

An Open Compiler using Meta-Level Information for Improving the Efficiency of Object-Oriented Systems

Tom Tourwé* and Wolfgang De Meuter
{Tom.Tourwe,wdmeuter}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050-Brussel-Belgium

Abstract

In the past few years, the discipline of object-oriented software reuse has received significant attention. Software developers strive to make a system comply to many non-functional requirements, such as maintainability, adaptability and composability, in order to be able to reuse their code in other systems. The pursuit of these goals however, results in systems that suffer from a significant performance loss, due to the many abstractions and the heavy reliance on late binding polymorphism. In this paper we will show that an open compiler using meta-level information about the design of a system can be used to alleviate this problem.

1 Introduction

Software reuse is still one of the “hot topics” of today’s research. It is generally acknowledged that object-oriented programming languages are a giant leap forward for the construction of such reusable systems. Properties such as inheritance and late binding polymorphism simplify the process of making a system comply to many important non-functional requirements, such as maintainability and adaptability. Popular techniques for achieving reusable systems, such as design patterns [GHJV95] and idioms [Cop92, Bec97], make extensive use of these properties and all share the same common goals:

- promote loose coupling between classes in order to be able to make them more interchangeable and independent of each other.
- use late binding polymorphism in order to provide support for future extensions

*Author financed with a doctoral grant from the *Instituut voor Wetenschap en Technologie*, Flanders

Unfortunately, the pursuit of these goals results in systems that are much less efficient. Because classes are loosely coupled, many more messages need to be sent between their objects in order to achieve the same behaviour. Also, due to the heavy use of late binding, the application of existing object-oriented compiler optimizations, such as inlining [ASU86], is prohibited because the compiler cannot predict at compile-time which method will be invoked at runtime. The result of all this is that compiled code remains largely unoptimized. Consider for example the *Visitor* design pattern from [GHJV95], which doubles the number of message-sends, because it makes use of the *double-dispatch* implementation technique. Another example is the *Decorator* pattern, which also doubles the number of message-sends since it constantly needs to redirect messages from a decorator object to the decorated object. Therefore, what we need is not just a smart compiler that reduces method-lookup time, but one that is able to *compile away* complete designs. Because compilers cannot possibly be that smart, we propose an open compiler in which the programmer can help the compiler by providing it with vital design information.

2 Current techniques to solve these problems

The problem stated in the previous section is hardly new and a number of techniques to alleviate it already exist. First, we will describe the support programming languages offer and argue why this support falls short. Second, we will discuss some more advanced techniques. Finally, we will indicate why each of these techniques still fail and will always do so.

2.1 Programming language support

Some programming languages, like C++ and Java for example, allow the programmer to specify whether a method should be subject to late binding or not by providing special keywords like `virtual` or `final`. The compiler can then make use of this information to statically bind the message-send and possibly inline the method.

This approach has some severe drawbacks however. First of all, it is unrealistic to let a programmer decide for each and every method in a system whether it needs to be overridden or not. Furthermore, future extensions may need to override methods in a way not foreseen in the original design. These observations clearly suggest that support in the programming language alone is not sufficient to solve the problems, even more so because the support they offer only allows local optimizations while we are more interested in architectural optimizations.

2.2 Advanced compiler techniques

Because programming language support turns out to be inadequate for solving the efficiency problem, more sophisticated techniques are needed. In order to optimize code significantly, a compiler should incorporate non-local information about a system. This should allow it to take less conservative decisions about the system's architecture when compiling. These observations

gave rise to techniques such as class hierarchy analysis and exhaustive class-testing [Dea96]. Although experiments show that incorporating these techniques in a compiler [DDG⁺96] are a serious improvement, we still believe they will not be able to optimize future systems significantly. In the next section, we will indicate why this is so.

2.3 Evaluation

The common factor among all techniques discussed above is that they try to predict at compile-time which particular method will be invoked at runtime. In order to do this, they try to infer the exact type an object will have at runtime at a particular call-site. Design patterns and idioms however, provide in essence a higher-order abstraction mechanism for solving certain problems in a particular context. Current optimization techniques are not able to eliminate these abstractions, since they do not know what these abstractions are, nor how they can be eliminated. Consider the *Visitor* and *Decorator* examples from the previous section. It is clear that type-prediction by itself will not be able to eliminate all extra message-sends, since the system consists of an entangled web of objects with a rather complex interaction between them. Current compilers have no architectural knowledge, so they cannot infer this information automatically. As a consequence, the resulting systems still suffer a vast performance loss.

3 Our approach: an open compiler using meta-level information

As a solution, we propose a transformational approach, much like refactoring [Opd92] but in the opposite direction: we want to transform code so that it becomes more efficient even if this means that it becomes less reusable. Of course, the programmer should always work with the reusable code and use the compiler to transform it into efficient code.

This approach is motivated by experiments we conducted, showing that most design patterns from [GHJV95] have a corresponding *efficiency pattern*. We define an efficiency pattern as a solution to an ever recurring problem in a particular context, just as a design pattern. The efficiency pattern's solution however does not focus on the non-functional requirements, but on efficiency. An efficiency pattern therefore sometimes corresponds to the antipattern [BMMM98] that is associated with the design pattern.

As it appears, it is relatively easy to automatically transform a design pattern into its corresponding efficiency pattern, thereby preserving the behavior of the system. To achieve this, we developed an open compiler, which is able to reason about object-oriented programs at the meta-level, and uses the information gathered in this way to perform the necessary transformations. A number of aspects need further elaboration: why do we need meta-level information, how can we provide the compiler with this information and why do we need an open compiler?

3.1 Why meta-level information?

Current compilers for object-oriented languages only have a (more or less) local view of the system. Due to this restriction they are obliged to take more conservative decisions and they cannot take into account global considerations. As already discussed, this results in a performance loss. If the compiler uses *structural* meta-level information, such as how all classes in the system are related to each other and how they collaborate in order to achieve certain behaviour, it has a more global view on the system as a whole. This allows it to easily recognize architectures and structures, such as design patterns, which are in essence nothing more but a set of classes interacting in a certain predefined way. Furthermore, if a decent meta-level interface is provided, the programmer can intervene and direct the compiler if this is needed.

An important advantage of making meta-level information explicit is that this allows the compiler to automatically transform recognized structures into more efficient ones. Of course, the programmer will need to specify how these structures should be transformed, so programmer intervention is required (this will be elaborated upon further in following sections). Although this can be regarded as a serious impediment (just as C++ and Java need programmer intervention for telling the compiler whether a method will be overridden), this need not necessarily be the case. For example, most design patterns have a corresponding efficiency pattern associated with them, as already explained. A library containing rules for transforming design patterns into efficiency patterns can thus be constructed and reused.

3.2 Why an open compiler?

There are several reasons why we want our optimizing compiler to be open. First of all, as explained above, it should be able to recognize certain structures and collaborations in a system and transform these into more efficient ones. Although some attempts are made [Bro97], it is generally very difficult to automatically detect design patterns in a system. This is mainly due to the fact that one particular pattern can be implemented in a variety of ways when used in different contexts. Clearly, the structure of a pattern should therefore not be hard-coded into the compiler. Since our compiler is open, the programmer is able to specify the structure of a specific design pattern.

For the very same reason, the transformation of a design pattern into its corresponding efficiency pattern should not be hard-coded. Since the implementation of a design pattern can vary between different systems, or even in the same system, it is clear that its transformation can also differ. Aside from specifying the structure of a certain design pattern, the programmer should thus also be able to specify the transformation of this pattern. This can only be supported by an open compiler.

Another important reason for using an open compiler is that new design patterns are discovered very frequently. As a consequence, an already existing compiler should be extensible and adaptable. That is, it should be able to incorporate information about these new patterns in an easy way, so that it can take this new information into consideration.

3.3 How to provide meta-level information to the compiler?

The last question we need to address is how we allow the programmer to specify which structures the compiler should recognize and how it needs to transform them. In our approach, the compiler is programmable at the meta-level by means of a logic language. Logic facts are used for expressing relations between different pieces of code, while logic rules are used to specify the transformations that need to be performed on this code.

We present this information to the compiler through the use of special comments (much like the `javadoc` comments). This has both the advantages that current compilers will still be able to compile the source (although they will not be able to optimize it significantly) and that the source code does not become cluttered with meta-level information code, which hampers readability, understandability and reusability.

4 Conclusion

In this paper, we proposed to use an open compiler using meta-level information to improve the efficiency of object-oriented systems. We showed that an optimizing compiler should use meta-level information because current techniques for improving the efficiency fall short as they concentrate only on statically binding message-sends and fail to incorporate global and architectural information of the system as a whole. Providing meta-level information to the compiler is a first step towards taking into consideration this more architectural view of a system.

Furthermore, we pointed out some reasons as to why we believe an optimizing compiler should be programmable at the meta-level and how this can be achieved elegantly by using a simple, expressive and declarative language. We intend to further investigate into this area to prove these claims valid.

References

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.
- [BMMM98] William Brown, Raphael Malveau, Hays W. III McCormick, and Thomas Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.
- [Bro97] Kyle Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, University of Illinois at Urbana-Champaign, 1997.
- [Cop92] James O. Coplien. *Advanced C++ programming styles and idioms*. Addison-Wesley Publishing Company, 1992.

- [DDG⁺96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex, an optimizing compiler for object-oriented languages. In *Proceedings of the OOPSLA 96 Conference*, pages 83–100. ACM Press, 1996.
- [Dea96] Jeffrey Adgate Dean. *Whole Program Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1996.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.
- [Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.