

Efficient Execution Model for Component-Oriented Software

ANDREAS GAL, PETER H. FRÖHLICH, AND MICHAEL FRANZ

Department of Information and Computer Science, University of California, Irvine, CA 92697-3430, USA

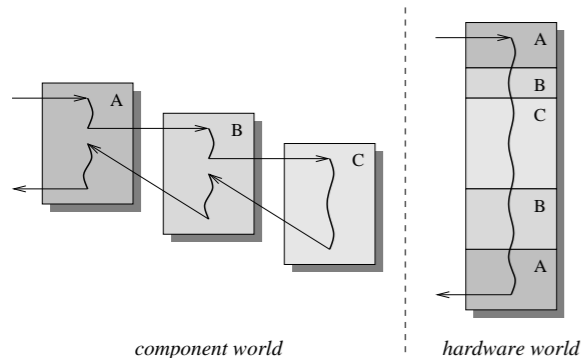
ABSTRACT

Traditional statically composed software can be globally optimized for execution speed already at compile-time. For dynamically composed component-oriented software a different execution model has to be employed to achieve similar performance. When components are compiled, little is known about the environment they will be used in later. Only after the deployment phase the required global information is available to aggressively optimize the overall system. This poster shows how runtime code recompilation can be employed to provide an efficient execution model for dynamically composed software.

1 Introduction

1.1 Motivation

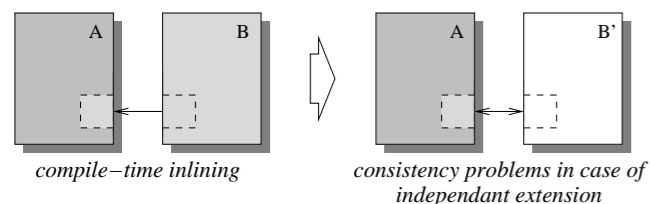
Our work on an efficient execution model for component-oriented software is motivated by a **fundamental mismatch of requirements** between component-oriented programming and the hardware layer, which is finally responsible to execute the component code.



The definition of a component - a unit of composition with contractually specified interfaces and explicit context dependencies only - demands, that **all code related to a component is contained within the component, and within that component only**. If components are composed without further global streamlining, every **inter-component message dispatch** will result in a **branch instruction at the execution level**. While this strict separation and distribution of the component code is required at the component level, it is very disadvantageous at the execution level. Modern pipelined processor architectures achieve much better performance if they can execute **linear code without many branch instructions**.

1.2 Optimizing Component Code

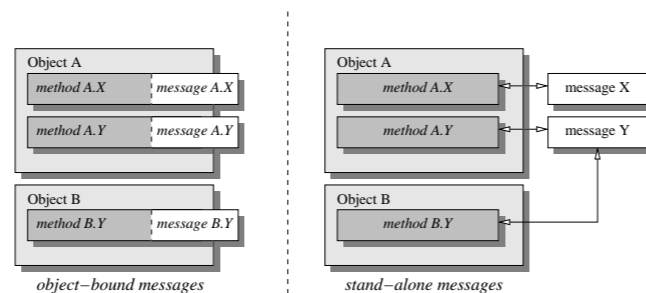
Even a simple optimization technique like inline expansion can have a great impact on the total execution time by reducing the number of branch instructions in the code. For components, inter-component inline expansion **cannot be performed at compile-time**. This would result in **inconsistencies in case of independent extension**.



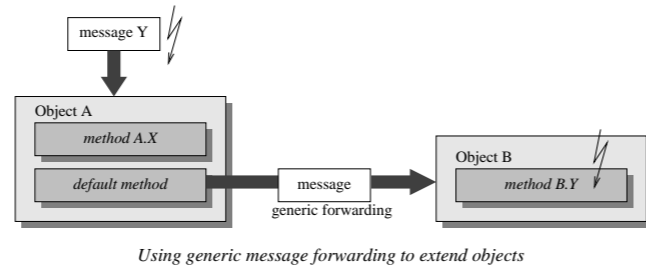
As components are traditionally shipped as compiled machine code, such simple optimizations are also **nearly impossible after the final composition**. To execute component-oriented software more efficiently, the binary component model has to be modified. The **native code generation and inline expansion is delayed** after the composition phase. To support this the component is shipped in some intermediate representation instead of native code. **Java and JIT compilers are the first step** in this direction. However, Java's Object Model has only **limited support for independent extensibility**.

1.3 Lagoona- A Component Language

Component Languages are specifically designed to allow for effective independent extensibility. Our research on efficient execution of component code execution is based on the component language Lagoona [1]. In contrast to most other object-oriented statically typed languages, in Lagoona **messages are stand-alone entities** and are not subordinated to classes.



The concept of inter-component structural inheritance causes many problems in the component-oriented domain. Stand-alone messages allow to use a **generic message forwarding** instead, which is much more suitable for component-oriented programming [3, 4].

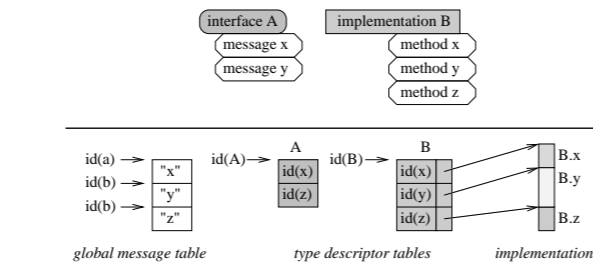


While being well suited for component-oriented programming, generic message forwarding further **increases the cost for an unoptimized message dispatch**. The remainder of this poster demonstrates how the message dispatch overhead can be drastically reduced by employing **runtime inline expansion and code recompilation**.

2 Message Dispatch

2.1 Object Model Runtime Representation

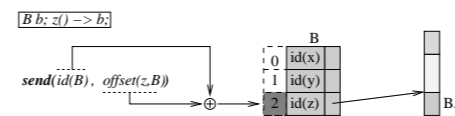
Lagoona supports two categories of types: **implementation types** refer to a concrete type, while **interface types** can refer at runtime to any implementation type which implements all messages required for that particular interface. At runtime, messages are represented in a **global message table** while for each type a **type descriptor** exists.



The global message table and the type descriptors are built at load-time and have to be updated if new components are loaded into the system or if a component is exchanged for a new version. While every inter-component message dispatch obviously requires late binding, Lagoona differentiates between two levels of late binding. The message dispatch to implementation types (**direct dispatch**) can be bound to a concrete method already ahead of time during the link process. The message dispatch through an interface however has to be resolved dynamically at runtime as the receiving method has to be determined by the real type of the object addressed by that particular interface reference.

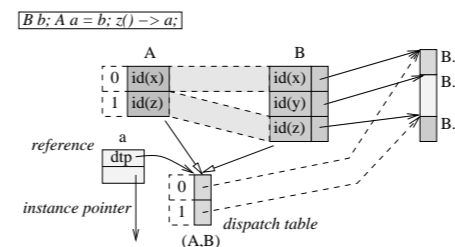
2.2 Direct Dispatch

A message dispatch operation to an implementation type is called a **direct dispatch**. Every direct dispatch can be **bound at link-time to a concrete receiving method** and thus can also be **inline expanded into the calling method** to eliminate the runtime cost of the message dispatch operation for this particular dispatch type all together. The figure below shows a message z sent to an implementation type B .



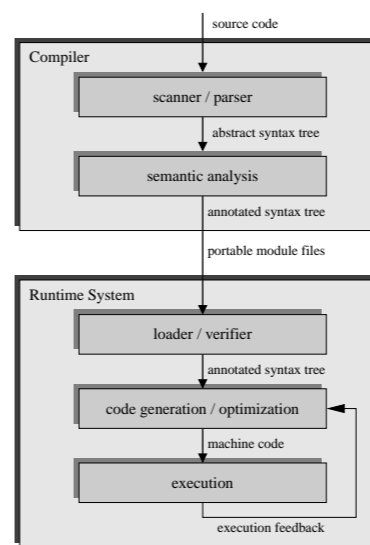
2.3 Indirect Dispatch

For an indirect dispatch, which is the case when a message is sent to an object through an interface, the **receiving method can only be determined at runtime**. In Lagoona we perform a **lazy creation of dispatch tables** each time an object is assigned to an interface reference. The **dispatch table maps the concrete methods of the object to the interface**. Recurring dispatch table requests are handled by a dispatch table cache.



3 Optimization

3.1 Runtime Recompilation



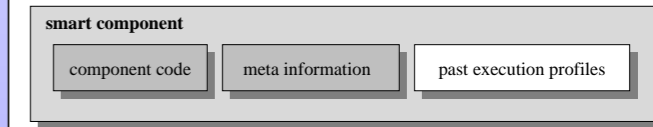
By design components often communicate with other components through interfaces. Thus, indirect message dispatch operations occur quite often in real-world applications. For these, the receiving method cannot be determined until a concrete object has been assigned to the interface reference. To increase the execution efficiency, we **dynamically recompile the code after the assignment has been evaluated**. At this point the receiving methods are known and optimization techniques like inline expansion can be applied. However, only the recompilation of often executed message dispatch operations pays off. The execution profile of the application has to be analyzed carefully to predict message dispatch operations which are highly likely to pay off after they have been recompiled.

3.2 Code Distribution Format

Many established component architectures like COM and CORBA distribute components in machine code form. Performing dynamic optimization as described by us on already compiled machine code is extremely difficult. Instead the code has to be transported to its final deployment environment in a more **high level format**. In our implementation we have decided to use an **annotated syntax tree** as code transport format [2]. It would have been possible to use byte code formats like Java or .NET byte code as well. However, we found that an annotated syntax tree can be transformed easier into a Static Single Assignment (SSA) form, which we are using as intermediate representation.

4 Current Implementation and Future Work

We have implemented a prototype Lagoona compiler and runtime system in Java. The prototype is able to compile and execute complex applications. We are currently in the process to re-implement the whole Lagoona framework in the Lagoona language itself to make it **fully self-hosted**. We use this process also to improve the internal design of the compiler. One area of future work we are especially interested in is to **improve the execution profile based pay-off prediction mechanism**, which is used to decide whether a particular indirect message dispatch operation is worthy a recompilation. Currently, each time the application is restarted, the execution profiling is started "from scratch". Instead, we want to feed back the execution profile information into the deployment unit. Thus, we turn it into a **smart component**, that adapts itself over time to its deployment environment.



5 Conclusions

Component-Oriented Programming limits the effectivity of traditional compile-time optimization techniques as components are independent units of deployment and composition. We proposed to delay the optimization process after the deployment process. Further, we described how execution profile information can be used to even optimize message dispatch operations which are not decidable ahead of time using dynamic re-compilation at runtime. To support this process, we proposed to use high-level representations of component code like annotated syntax trees instead of machine code and briefly presented our prototype implementation of a Lagoona compiler and runtime system.

References

- [1] M. Franz. The programming language Lagoona: A fresh look at object-orientation. *Software: Concepts and Tools*, 18(1):14-26, Mar. 1997.
- [2] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87-94, Dec. 1997.
- [3] P. H. Fröhlich and M. Franz. Stand-alone messages: A step towards component-oriented programming languages. In J. Gutknecht and W. Weck, editors, *Proceedings of the Joint Modular Languages Conference*, volume 1897 of *Lecture Notes in Computer Science*, pages 90-103, Zürich, Switzerland, Sept. 2000. Springer-Verlag.
- [4] P. H. Fröhlich, A. Gal, and M. Franz. On Reconciling Objects, Components, and Efficiency in Programming Languages. Technical Report 02-12, Department of Information and Computer Science, University of California, Irvine, Mar. 2002.

This work was partially supported by the National Science Foundation under grants EIA-9975053 and CCR-0105710.