An Efficient Execution Model for Dynamically Reconfigurable Component Software[*]

Position Paper

Andreas Gal gal@uci.edu
Peter H. Fröhlich phf@acm.org
Michael Franz franz@uci.edu

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

Traditional, statically composed software can be globally optimized for execution speed at compile-time. For dynamically composed component software, a different execution model has to be employed to achieve comparable performance. As components are developed and compiled separately, little is known at compile-time about the environment they will be used in later. The global information required to aggressively optimize the overall system is only available after the deployment phase. In this paper, we discuss how dynamic recompilation can be employed to provide an efficient execution model for dynamically composed software.

1 Motivation

While component-oriented programming [19] is by now widely accepted in the academic world, the majority of commercial software is still implemented in a monolithic fashion. Many developers treat component-oriented programming with reserve because of the inherent performance problems that many component-oriented languages and frameworks exhibit. These performance problems are not incidental, but stem from a fundamental mismatch between component-oriented programming and the hardware layer, which is ultimately responsible for executing the composed 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

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

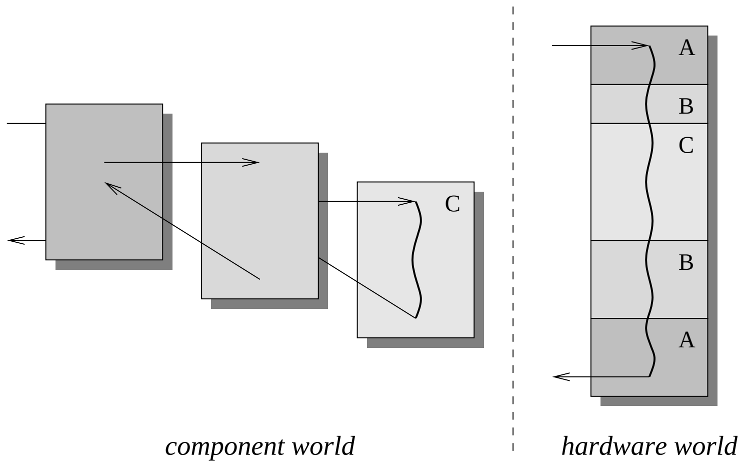

Figure 1: Conflict of interests between components and the hardware layer.

component only. If components are composed without further global streamlining, all inter-component communication will result in a branch instruction on the hardware layer. While the strict separation and encapsulation of the 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 executed linear code without many branch instructions (Figure 1).

The remainder of this paper is organized as follows. In Section 2 we describe simple optimization techniques which can be applied to reduce the inter-component communication overhead, and discuss why many of these techniques cannot be applied at compile-time for component software. In Section 3 we propose an execution model for component software based on dynamic recompilation, which allows to apply inter-component communication optimization techniques even in environments which are subject to runtime recomposition. Our prototype implementation of an aggressively optimizing component-oriented programming framework is described in Section 4. In Section 5 we discuss related work and Section 6 contains conclusions and describes ideas for future work.

2 Optimizing Component Code

The strict encapsulation of code into components makes it impossible to apply common optimization techniques like *inline expansion* [14] over component boundaries. Inline expansion is an optimization technique which replaces frequently performed message send operations with the body of the method triggered by the reception of that message on the receiver end. The technique has been applied successfully to compiling statically composed object-oriented programming languages [2]. As a side effect, inline expansion often increases the average basic block size, which can lead to better results for some related optimization techniques like dead-code elimination and constant propagation [15].

In case of component-oriented programming, most of these optimization techniques are limited to intra-component message sends. The earliest point when global optimization of inter-component communication can be performed is composition time. In a system with purely static composition—where components are loaded when the application is started, but never exchanged—this global optimization process can be performed ahead of execution. However, it still has to be performed in the deployment context and cannot be done by the component producer. In a dynamic composition environment, where components are loaded, unloaded, exchanged, or updated at runtime, the global optimization of inter-component communication cannot be performed ahead of time. Instead, it has to be performed just-in-time when a new component is loaded into the system.

3 Dynamic Recompilation

In the previous section we have argued that it is required to shift the global optimization process into the deployment environment to allow aggressive optimization of inter-component communication. This means that the traditional "unified" compiler, which takes source code as input and generates machine code as output, has to be split into (at least) two parts: the backend part must be relocated into the deployment environment.

Separating the compiler frontend from the code generating backend is common in many modern language infrastructures, for example in Java [12] and .NET [4], although the original motivation was portability rather than dynamic software composition. Existing just-in-time compilers for these platforms are often unable to cope with the requirements created by component-oriented programming. To support independent extensibility, component communication usually occurs using abstract interface types rather then concrete implementation types. At component load-time, the actual implementation type of an object is not necessarily visible through the interface reference used to denote it. Therefore, the actual receiving methods of message sends through that interface reference are not known either. Instead, the correct target method will be selected at runtime depending on the concrete implementation type. This approach is also referred to as *late binding*.

In lieu of compiling the source code directly to machine code for some target architecture, an intermediate representation is generated. To perform inline expansion of message sends which use late binding, we apply dynamic recompilation [6] in our execution model. The runtime system contains a code generator which translates this intermediate representation into machine code at load-time (Figure 2).

However, this translation process is not limited to load-time. To deal with

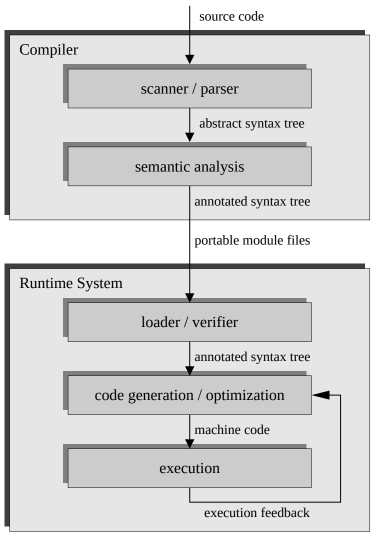

Figure 2: Global optimization using dynamic recompilation.

late binding, the execution engine tracks assignments to interface references and triggers the inline expansion as soon as the definitive target method is known. While most existing dynamic compiler implementations like the Sun HotSpot Virtual Machine for Java [18] assume that message dispatch through an interface is less likely than early binding, we assume that interface calls are actually the common case. In case of COP this is appropriate as components heavily use interfaces to abstract inter-component communication and to increase reusability. In contrast to HotSpot we allow multiple compilations of the same intermediate code to co-exist in memory. Depending on the actual interface call receiver the corresponding trace is executed.

The result is a *dynamic compilation* process which performs *continuous program optimization* [16, 17]. As the recompilation process has a significant cost itself, the dynamic code optimizer has to decide on-the-fly whether a possible optimization it detected is likely to pay off at some point in the future or not. Only optimizations which promise to be profitable are actually performed.

4 Implementation

We have implemented the proposed execution model as part of the Lagoona [5, 8] component-oriented programming framework [10]. Lagoona is a programming language explicitly designed for component-oriented programming. It is based on the ideas of *stand-alone messages* and *generic message forwarding* [9]. Our current prototype implementation uses an annotated syntax tree [7] as the code transport format at the component level. It would have been possible to use lower-level byte-code formats like Java or .NET as well. However, we found that an annotated syntax tree can be transformed into Static Single Assignment form [1] more easily, which greatly simplifies many common code optimization algorithms.

5 Related Work

The term *dynamic compilation* has originally been coined in the context of Smalltalk-80 [3]. The Smalltalk-80 compiler emitted portable *v-code*, which in turn was translated to native machine code (*n-code*). Self [20] is a dynamically typed, object-oriented language, which in its later incarnations Self-91 and Self-93 [13] also used profiling and dynamic compilation to speed up performance-critical parts of the program. However, the approach undertaken by Self significantly differs from our execution model, as Self lacks explicit support for component-oriented programming. An interesting feature of some Self implementations is to do dynamic "de-optimization" to assist debugging. We are considering to add this feature to Lag*oo*na. Earlier work in the area of efficient light-weight component architecture includes Open Components [11]. Open Components eliminate all inter-component communication overhead using compile-time code manipulation. However, Open Components—by design—do not support dynamic composition. Our dynamic compilation approach also relates closely to work done by Kistler on continuous program optimization [16, 17].

6 Conclusions and Future Work

In this paper, we presented dynamic recompilation at runtime as a means to overcome inherent performance problems in current component-oriented programming frameworks. We identified the central ideas of component-oriented programming—independent and dynamic extensibility—as the major obstacles that limit the effectiveness of traditional compile-time optimization techniques. We argued for delaying the optimization until after the deployment process to allow for the global optimization of inter-component communication. Furthermore, we proposed to use dynamic recompilation to optimize even method calls going through interfaces, which are not optimizeable ahead of time. We pointed out that many currently existing just-in-time compilers are not designed to perform inline expansion of interface calls, which has a significant impact on the execution performance of component-oriented software. As far as future work is concerned, we have identified the profitability prediction in our current implementation as one of major weaknesses. Our prototype uses a very simplistic heuristic to decide whether a possible optimization should really be performed. Preliminary measurements have shown that the prototype often spends a considerable amount of time to recompile code fragments, which in the end fail to deliver the speedup the predictor has been hoping for. We plan to focus our future research on more advanced prediction mechanisms to ensure a higher rate of profitability for the overall optimization process.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in progams. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, Jan. 1988.

[2] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In W. Olthoff, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, Aug. 1995. Springer Verlag.

[3] L. P. Deutsch and A. M. Schiffmann. Efficient implementation of the Smalltalk-80 System. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 297–302, Salt Lake City, UT, Jan. 1984.

[4] ECMA. The .NET Common Language Infrastructure. Technical Report TR/84, ECMA, Geneva, Switzerland, June 2001.

[5] M. Franz. The programming language Lagoona: A fresh look at object-orientation. *Software: Concepts and Tools*, 18(1):14–26, Mar. 1997.

[6] M. Franz. Toward an Execution Model for Component Software. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP)*, pages 144–149, Mar. 1997.

[7] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.

[8] 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.

[9] P. H. Fröhlich and M. Franz. On certain basic properties of component-oriented programming languages. In D. H. Lorenz and V. C. Sreedhar, editors, *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 15–18, Tampa Bay, FL, Oct. 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[10] 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.

[11] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. Open Components. In *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Tampa, Florida, Oct. 2001.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, 2nd edition, 2000.

[13] U. Hölzle. *Reconciling High Performance with Exploratory Programming.* PhD thesis, Department of Computer Science, Standford University, 1994.

[14] W.-M. W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. In *Proceedings of SIGPLAN 89 Conference on Programming Language Design and Implementation*, Portland, OR, 1989.

[15] S. Jinturkar. *Data-Specific Optimizations.* PhD thesis, Department of Computer Science, University of Virginia, May 1996.

[16] T. Kistler. *Continuous Program Optimization.* PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, Nov. 1999.

[17] T. Kistler and M. Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[18] Sun Microsystems. The Java HotSpot Virtual Machine: Technical White Paper, 2001.

[19] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[20] D. Ungar and R. B. Smith. SELF: The Power of Simplicity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242, Orlando, FL, Dec. 1987.