Architecture-Based Program Compaction

Chris Lüer André van der Hoek

School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{chl,andre}@ics.uci.edu

Recently, software for systems with small memories has become a focus of research [3]. Mobile networked devices, ubiquitous computing, embedded systems are some of the trends that increase the need for software that takes up very little space — the less space, the better.

One aspect of small memory software is the need for smaller programs; i.e. reducing the size of the program code itself. In many systems, a large part of the memory is taken up by the program code, and not by objects created at runtime. Hence, program compaction [1] is considered a useful technique for saving memory space. In this paper, we propose a new approach to program compaction for applications that contain reused components. We call our approach *architecture-based*, since it makes use of designer's understanding of the architecture of an application that consists out of reused components.

Current approaches to program compaction are either fully automatic or fully manual. Automatic approaches include code compression and extraction techniques. Compression [4] compresses the program using common data compression techniques, and tries to minimize meta-information stored in the executable file such as fully qualified names of external classes; extraction [5] uses static program analysis in order to remove unreachable code and unused declarations. Automatic approaches work without interaction by a designer, and so they cannot take advantage of knowledge about the architecture of a program.

Manual approaches to program compaction require designers to analyze the source code for code that is redundant or of low importance, and to manually remove this code.

These approaches are not satisfactory when dealing with reused code, such as third-party libraries. Manual code compaction is not possible because (1) there is insufficient knowledge about the library to modify its design; and (2) the source of the library may not be available. Automatic code compaction is equally insufficient, because it cannot be aware of design considerations and thus cannot compact code optimally. Since reusable components often include much more code than needed by a given application, they are an important target for reducing code bloat.

We believe that tools are needed that can reconfigure reusable components based on an application designer's needs. Automatic compression and extraction tools can, in a first step, remove library code that is never called and other obviously redundant information; but in a second step, a designer needs to be enabled to exchange memory-consuming parts of the library with more space-conserving implementations. While extraction tools only allow to choose between including a subcomponent, or completely removing it, we believe that smarter tools should allow a subcomponent to be replaced by a more efficient one.

Such a tool will give the designer a way to reconfigure existing code by providing alternative implementations for arbitrary parts of a component. The designer should be able to perform the following steps:

1. Identify subcomponents that are inefficiently written, or not optimized towards the current application. In a Java program, a suitable granularity for subcomponents might be class files.
2. Isolate the subcomponent and identify its dependencies. The designer needs to be able to find out how the subcomponent interacts with the rest of the library, so that it can be replaced.
3. Take out the subcomponent, and replace it by a more efficient implementation.

Example situations in which such a tool might be used include:

- A reused component A depends on another large component B. Component C has a very similar

functionality to B, but is much smaller. The proposed tool could be used to reconfigure A so that it uses C (or an adapted version of C) instead of B.

- A reused component A depends on many other components, which have some overlap of functionality. Reconfigure A so that it uses a newly written component B that provides the services needed by A in a more compact form.
- A number of components implement rarely-used functionality. Reconfigure the components using them to deploy more generic, more compact code instead. In most systems, a large percentage of code is taken up by the handling of rare special cases and exceptions. Significant savings in code size are possible if these functions are replaced; for example, a specific error message may be replaced by a more generic one, or a rare kind of input may be ignored.

We are currently developing a tool that allows for reconfiguration of legacy Java components in the absence of source code. The tool is intended to solve a number of related deployment-time problems that occur in the context of reuse of third-party components. An example of the use of this tool not related to code compaction is resolving name conflicts. Name conflicts between deployed Java components are a common issue [2]; the proposed tool will be able to rename identifiers in deployed components so that name conflicts disappear.

We define *reconfiguration* as the activities of changing dependencies and adding adaptations to existing code. *Changing dependencies* means that if a component A uses a component B, the tool can force A to use C instead of B. By *adaptations*, we mean adapting an interface of a component so that it better fits its requirements. Of course, the facility of changing dependencies allows us to replace a component by a wrapper that delegates its calls to the original components; hence, adaptation support is somewhat optional.

To make reconfiguration possible, the proposed tool provides an extended Java class loader. The class loader is a part of the Java platform that is used to load classes from the hard disk into memory and to dynamically link them. The extended class loader analyzes classes before they are loaded, and applies modifications to them, thus enforcing reconfigurations. Since the class loader is an integral part of the platform, this approach works with all legacy code independent of its architecture, with the exception of code using reflection (meta-programming). A second advantage of the class loader approach to reconfiguration is the fact that the compiled class files need not be modified; all modifications can happen transparently at run-time. The performance overhead of dynamic reconfiguration should be negligible in most cases since class loading is already a very expensive activity.

The extended class loader uses configuration files to determine the desired modifications. A configuration file is a text file specifying the desired dependency changes and adaptations. Based on an understanding of the architecture of an application, a designer can override individual parts of the architecture by specifying dependency changes and adaptations in such a file.

Summary. We outlined the need for reconfiguration tools in code compaction. Current compaction techniques are either fully manual or fully automatic, but there is a lack of compaction techniques that are driven by architectural understanding. We are proposing a tool based on an extension of the Java class loader that can be used to reconfigure legacy Java code. Reconfigurations can be used to change the architecture of a system at deployment time, and thus can be used to replace large reused components by smaller, equivalent ones.

References

1. De Sutter, B. and De Bosschere, K. Software Techniques for Program Compaction. *Communications of the ACM, 46*, 8 (August 2003). 33-34.
2. Hnětynka, P. and Tůma, P. Fighting Class Name Clashes in Java Component Systems. In *Proceedings JMLC 2003*, Springer, 2003.
3. Noble, J. and Weir, C. *Small Memory Software*. Pearson Education, Harlow, 2001.
4. Pugh, W. Compressing Java Class Files. In *SIGPLAN '99*, 1999, 247-258.
5. Tip, F., Sweeney, P.F. and Laffra, C. Extracting Library-Based Java Applications. *Communications of the ACM*, *46,* 8 (August 2003). 35-40.