UC Irvine
UC Irvine Previously Published Works

Title

Adaptive Techniques for Minimizing Middleware Memory Footprint for Distributed, Real-Time, Embedded Systems

Permalink
https://escholarship.org/uc/item/66b0g4fr

Journal
Proceedings of the IEEE 18th Annual Workshop on Computer Communications, 18

Authors
Panahi, Mark
Harmon, Trevor
Klefstad, Raymond

Publication Date
2003-10-20

DOI
10.1109/CCW.2003.1240790

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Copyright © 2003 IEEE. Reprinted from the *Proceedings of the IEEE 18th Annual Workshop on Computer Communications* (CCW 2003).

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of California eScholarship Repository's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or re-distribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

Adaptive Techniques for Minimizing Middleware Memory Footprint for Distributed, Real-Time, Embedded Systems

Mark Panahi, Trevor Harmon, and Raymond Klefstad
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, California
{mpanahi, tharmon, klefstad}@uci.edu

***Abstract*—In order for middleware to be widely useful for distributed, real-time, and embedded systems, it should provide a full set of services and be easily customizable to meet the memory footprint limitations of embedded systems. In this paper, we examine a variety of techniques used to reduce memory footprint in middleware. We found that combining aspect-oriented programming with code shrinkers and obfuscators reduces the memory footprint of CORBA middleware to <5% of its original size, as customized for a small client application for a memory-constrained embedded device.**

Keywords-distributed systems; embedded systems; real-time systems; CORBA; middleware; aspect-oriented programming; memory footprint

I. Introduction

Middleware platforms, such as J2EE [1], CORBA [2], and .NET [3], are widely used in many applications, because they provide a broad set of frequently used capabilities that save time and simplify the development process. Essentially, the use of middleware allows the developer to avoid having to reinvent custom solutions for each set of applications.

Developers of embedded systems would be able to derive tremendous benefits from middleware usage, except that most middleware implementations are too large to fit on resource-limited embedded devices. Therefore, in order for the middleware development community to address the resource constraints of embedded devices, the size of the middleware implementation must be customizable to accommodate the constraints imposed by the underlying device. At the same time, this customization must be achieved easily by the application developer, in keeping with the spirit and goals of middleware.

A variety of techniques can be applied to reduce the memory footprint of middleware code, including the following:

- Conditional compilation,
- Java reflection combined with dynamic class loading,
- Code shrinking,
- Code obfuscation, and
- Aspect-oriented programming (AOP) [4].

The emphasis of this paper is on the advantages of the systematic use of AOP in conjunction with code shrinking and obfuscation.

The remainder of this paper is organized as follows: Section II gives an overview of techniques for reducing memory footprint; Section III presents the uses of AOP for reducing memory footprint in ZEN [5], a full-featured Real-time CORBA Object Request Broker (ORB) for RTSJ (Real-time Java) [13]; Section IV presents empirical results when AOP, in conjunction with code shrinking and obfuscation, was applied to ZEN; Section V presents our conclusion; and Section VI describes related work.

II. Approaches to Reduce Memory Footprint

A. Conditional Compilation

With conditional compilation, a preprocessor examines compile-time configuration flags to decide which portions of the code to include or remove, as defined by the application programmer (with `#ifdef`, for example). Conditional compilation is often used with C or C++ to allow retargeting of systems to different platforms. It has also been used to allow custom configuration of operating systems and middleware [6]. A disadvantage of this technique is that the resulting source code is difficult to read and debug. In addition, the application developer must be aware of the wide variety of compile-time flags that must be specified for each compilation. Furthermore, this approach is unsupported in and undesirable for Java.

B. Reflection and Dynamic Class Loading

With reflection and dynamic class loading, an ORB can be designed with a high degree of pluggability, allowing unused features to be eliminated from the ORB core and loaded dynamically only when needed. The Virtual Component Pattern was used extensively in the first version of ZEN to allow automatic subsetting of middleware features [5]. This pattern, which behaves like an object-oriented virtual memory, is particularly useful when a middleware feature may have a variety of implementations and only one is used at a particular time. The unused implementations may be eliminated from

This work was funded by Boeing, DARPA, and AFOSR.

memory, thus reducing the memory footprint. However, a disadvantage of this technique is that dynamic class loading, occurring at run-time, can cause jitter that is unacceptable for some systems.

C. Code Shrinkers

Code shrinkers use a variety of techniques for shrinking code. Basic implementations walk through a list of classes and remove any fields that are declared but never accessed. For example, a field declared in a class as private, static, and final, but never used within that class, is unnecessary and can simply be removed. More advanced code shrinkers build a call graph of all possible execution paths that the code could take at run-time. Any classes and methods not in the graph are removed, as shown in Figs. 1 and 2.

D. Code Obfuscators

Reverse engineering of software has traditionally been difficult due to the low-level nature of compiled code. With Java, however, reverse engineering is surprisingly easy, due to its high-level instruction set known as "bytecode". Bytecodes, which are translated at run-time into the platform's native instruction set, include the full names of methods, packages, and variables, even after all debugging information has been removed. Furthermore, bytecodes are well specified and have a nearly one-to-one mapping with Java language constructs [7]. For these reasons, changing bytecodes back into a near-perfect representation of their original source code is relatively simple. A number of tools, known collectively as "decompilers," are available to decompile bytecodes back into Java source code.

To hinder reverse engineering using these decompilers, Java developers often apply a tool known as an "obfuscator" to their compiled code. The obfuscator changes the bytecodes so that meaningful method names, such as `getBufferSize()`, become cryptic names such as `hjk()`. This renaming is also known as *name mangling*. Although decompilers can still reverse-engineer obfuscated bytecodes, the meaningless names greatly reduce the value of the generated source code. The name mangling also has an interesting side effect: after obfuscation, when verbose, descriptive names are converted to shorter, simpler names, the total size of the compiled code shrinks.

Many obfuscators take the idea of code shrinking several steps further. During the obfuscation process, they search for unused classes, methods, and fields, and remove them entirely from the bytecode. For applications that use only a small part of a shared library, the reduction in size can be dramatic.

E. Aspect-oriented Programming

Aspect-oriented programming is a relatively new programming methodology that gives programmers the ability to refactor and modularize crosscutting concerns resistant to modularization by traditional object-oriented techniques. Many of the features in middleware are represented by such crosscutting concerns and thus can be excluded easily from a given build process by the application developer via the use of AOP.

```
public class Hello {
public Hello(boolean b) {
  if (b)
    methodA();
  else
    methodB();
}

private void methodA() {
  System.out.println("Method A");
}

private void methodB() {
  System.out.println("Method B");
}

private void methodC() {
  System.out.println("Method C");
}

public static void main(String[] args)
{
  Hello h = new Hello(args.length > 0);
}
}
```

Figure 1. The typical Java compiler will compile the unused method C, but code shrinkers will remove it.

Modularization is an important element of software customization and memory footprint reduction. The ability to capture all of the mechanics of a given feature within a single module, including interactions with other modules, provides greater control over the inclusion of that feature in the main corpus of code. This is especially useful for an application to be executed on a memory-constrained device using a feature-rich middleware platform such as CORBA. The underlying middleware implementation must be customized to minimize

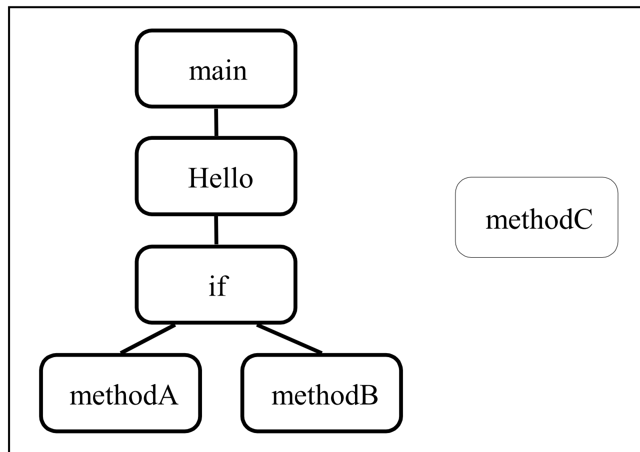

Figure 2. Code shrinkers use call graphs, such as this graph of the code in Fig. 1, to conclude that method C is never called and can be removed.

the presence of unneeded code (usually in the form of superfluous features).

Standard object-oriented programming techniques, primarily performed by representing logical entities as *classes*, are sufficient for modularizing most features in middleware. However, they lack the ability to modularize mechanisms that represent a single feature, but are scattered throughout several classes. Aspect oriented programming provides this facility.

III. Aspect-oriented Programming with AspectJ

AspectJ is an aspect-oriented programming language extension to Java [9]. It has been successfully applied to ZEN to achieve footprint reduction via the modularization of crosscutting concerns. It is important to note that there are two basic mechanisms of crosscutting. *Static* crosscutting allows the addition of methods and fields to classes. Subsetting via static crosscutting thus produces a net effect similar to that of code shrinking, but code shrinking accomplishes this type of code reduction much more easily. We therefore used the code shrinking technique, instead, to accomplish this reduction. On the other hand, *dynamic* crosscutting enables the addition of instructions (i.e., lines of code) to existing methods. This mechanism therefore allows the customization of run-time execution, at compile time, when it is known beforehand that certain code will not be needed. This customization cannot be accomplished through code shrinking alone and therefore provides an important mechanism for additional code reduction. In particular, this reduction is possible because AspectJ specifies well-defined locations in Java code, called *join points*, where additional code can be *woven in* (such as method calls and field references).

We identified two crosscutting concerns to exclude from ZEN's client ORB:

1. Support for local invocations: One of the main features of CORBA is to provide location transparency. However, if it is known ahead of time that all invocations are remote, then the code associated with local invocations (a large part of the server code) can be removed.

2. Portable interceptors: Portable interceptors provide hooks into the ORB code so that applications or services can examine requests as they are being processed (e.g., for authentication) on both the client and the server. Much of the code associated with portable interceptors is scattered throughout the ORB, and is thus a feature that can be properly modularized only with AOP. Fig. 3 shows the degree of code scattering formerly associated with portable interceptors, and indicates the exact locations where portable interceptor code has been woven in. It is also important to note that there are additional files not shown that are a part of the portable interceptor feature, but are referenced by the woven-in code. Thus, code footprint reduction results not only from the exclusion of woven-in code, but also from the exclusion of code upon which the woven-in code depends.

IV. Empirical Results

One of our research goals for ZEN is to enable the reduction of its code size so that it can fit a wide variety of small-footprint, network-aware, embedded systems for which it was intended. Furthermore, this code reduction, via customization, should be as easy as possible for the application developer. One step toward this goal is to build a customization framework that will use combinations of code reduction techniques to allow the size of ZEN to be decreased according to the requirements of a particular memory-constrained device upon which it would be installed. As a reasonable goal for target code size, we refer to the Java 2 Platform Micro Edition (J2ME) Connected Device Limited Configuration (CLDC), which is targeted for devices that typically have 128 KB to 512 KB of memory available for the Java platform and applications [8]. Because the target code size for these embedded devices is so small, even small reductions in code size become highly significant.

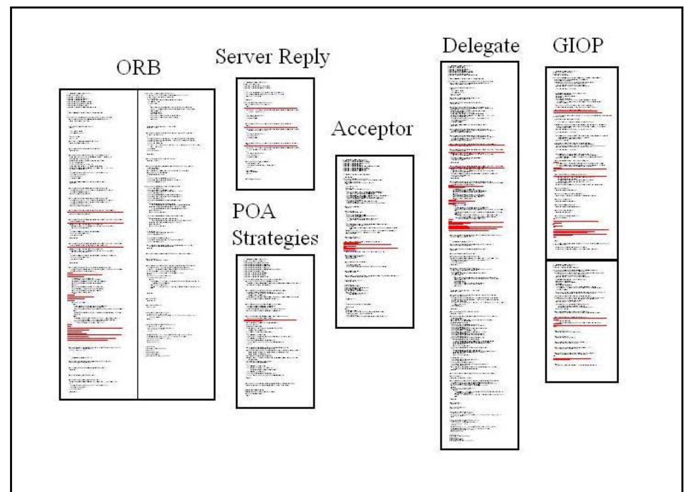

Figure 3. Portable interceptors are a single feature, but its code is scattered over many files. Portable interceptor code itself is shown in red.

In this work, we measured the ability of three of the techniques discussed above - AOP, code shrinking, and code obfuscation - to reduce the code size of ZEN. AOP has the unique ability to modularize crosscutting concerns, thus making those features pluggable. Because this modularity allows code shrinkers and obfuscators to exclude unneeded modules during the build process, we expect AOP used in conjunction with the other two techniques to yield synergistic code size reduction.

We used AspectJ, in conjunction with ProGuard [10], a Java code shrinker and obfuscator, to reduce the static memory footprint of ZEN so that only the code needed to service invocations of a client involving primitive data types is included. (This application example was chosen to represent the limited needs of a number of relatively simple embedded systems.) We then measured code size with the "aspectized" features either included or excluded, before and after code shrinking and obfuscation, and before and after the application of all three techniques.

Subsetting using aspects provides a modest reduction of code size of 270 KB, or 6% (see Fig. 4). This modest reduction is to be expected. We applied only dynamic crosscutting, relying on code shrinking to achieve (more easily) the

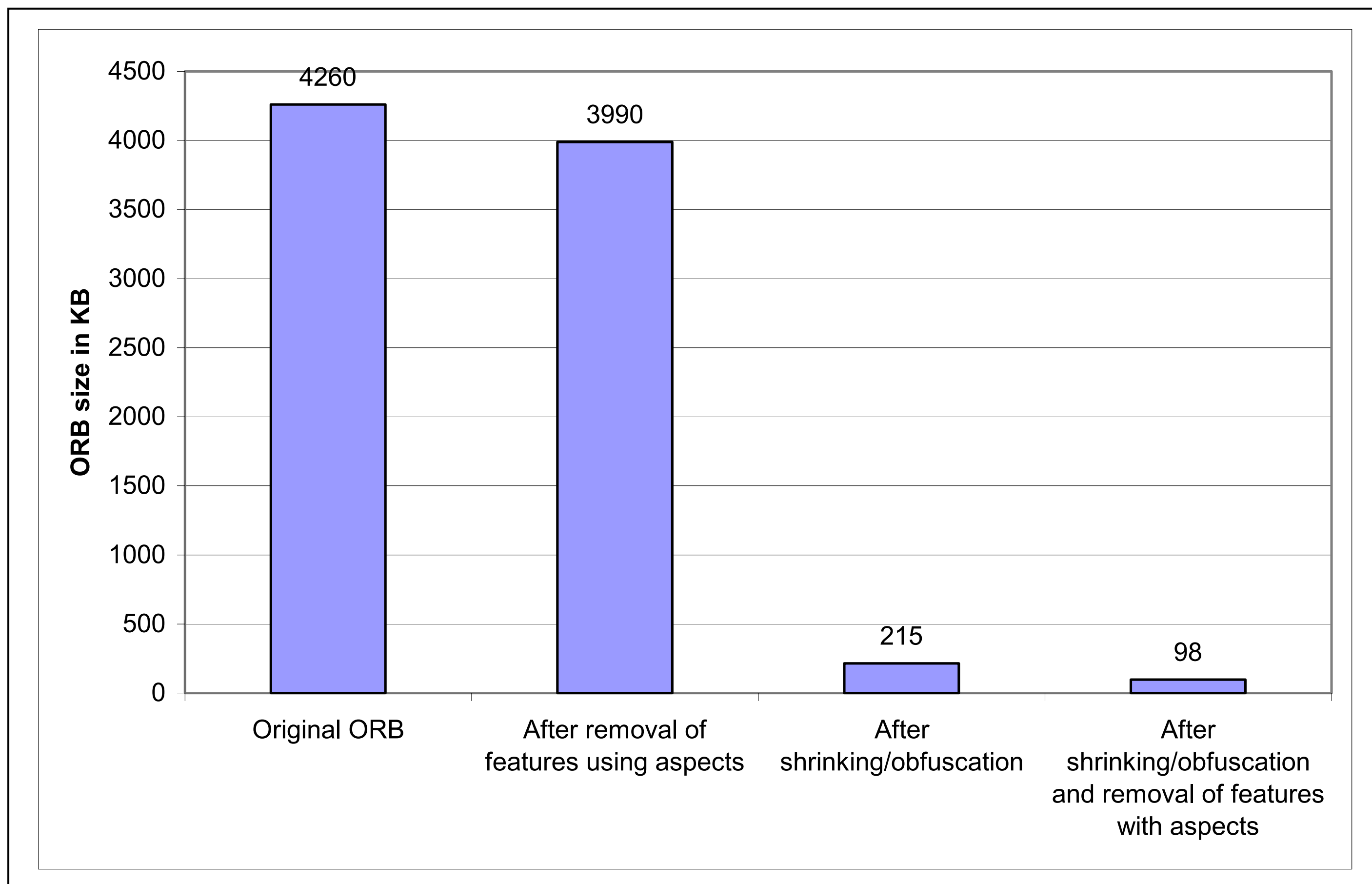

Figure 4. Memory footprint savings (in KB) with three code reduction techniques.

reduction that static crosscutting could also achieve. Therefore, the reduction seen using dynamic crosscutting represents only that which cannot be achieved through any other techniques. Furthermore, for this paper we identified only two areas with crosscutting concerns to serve as examples of those that could be addressed by aspects; to fully realize the benefits of aspects, more such areas can be identified.

Since code shrinking and obfuscation are related techniques, which are not independent in their application with the same tool, we primarily consider the effects of both of these techniques used together. Code shrinking and obfuscation together yield a dramatic reduction in code size of 4045 KB (95%), from over 4260 KB to 215 KB (see Fig. 4). This dramatic code reduction is also to be expected, since in this scenario, the shrinking and obfuscation reduce the ORB to one that provides only a basic set of features needed by an application. ZEN, like any full implementation of CORBA, contains code to support a large number of features, most of which are not needed for basic operations. While it is important for the ORB to provide a full set of CORBA services to meet the needs of a wide variety of applications, any given application will typically need only a subset of these services [5]. Shrinking, therefore, can yield dramatic reductions by removing most of this unnecessary code, while some additional reduction is derived from obfuscation due to the shortening of remaining field, method, class, and package names.

In addition, our results confirm that these three techniques work together in complementary ways to reduce code size dramatically. Although each of the three techniques can yield definite benefits individually, they can be combined synergistically to further reduce code size. The first step toward reducing ORB code size is to use aspects to achieve a higher degree of modularity in the ORB. The greater the degree of modularity, the greater the effectiveness the second step, code shrinking, will have. Code shrinking removes unneeded code, while aspects makes more of the unneeded code removable by eliminating some of the coupling with needed code.

For example, the aspect that represents support for local invocations is implemented in such a way that code shrinking is required for it to be effective. There is no discernible difference between the pre-shrunk versions of the full-featured ORB and the ORB with local invocation support excluded, because this aspect in fact eliminates only *dependencies* to modules used for local invocations. Removal of these dependencies, however, enables the shrinker to remove unneeded classes and methods associated with local invocations. Consequently, the version with local invocation support excluded is 3258 KB (76%) smaller than the full-featured version (see Fig. 4). The shrinker has been able to remove code that the aspect has, in effect, released.

Finally, code obfuscation, applied to the remaining minimal needed code, provides additional reduction that works

0-7803-8239-0/03/$17.00 ©2003 IEEE.

particularly well with AspectJ. We have observed that AspectJ tends to create methods with unusually long names. Obfuscation is particularly helpful in counteracting this side effect of using AspectJ.

Using aspects together with shrinking and obfuscation brings the code size of ZEN within the target memory requirements of an embedded device (for example, the 128-512 KB target requirements for the J2ME CLDC). Shrinking and obfuscation dramatically reduce the code size to a manageable level, within the maximum for the J2ME CLDC. However, the additional and synergistic reduction seen when applying code reduction and obfuscation to aspect-excluded code (i.e., from 215 KB to 98 KB) is important: this reduction makes the use of the ORB possible for some very small memory-constrained embedded devices.

V. CONCLUSION

Our initial tests have shown that a simple middleware implementation can be shrunk statically to less than 5% (e.g., from 4260 KB to 98 KB) of its original, un-optimized size. Furthermore, this reduction can be achieved easily and, in part, automatically by the application programmer (in contrast to, for example, using conditional compilation). In future work, we plan to explore the use of these techniques with more sophisticated examples, and to explore the dynamic memory footprint. The example measured in this paper, meanwhile, suggests that using this combination of code reduction techniques is a promising approach toward reducing ORB size for memory-constrained embedded devices.

VI. RELATED WORK

There has been other research related to using AOP in CORBA middleware. In [11], the authors successfully modularized several CORBA crosscutting concerns, including portable interceptors, the dynamic programming model (the Dynamic Invocation Interface and the Dynamic Skeleton Interface), and fault tolerance support, with AOP (via AspectJ). The authors proved the efficacy of applying AOP to CORBA by retrofitting an existing open-source implementation of CORBA (in this case ORBacus) with aspectized versions of these features.

In [12], the authors developed the Framework for Aspect Composition for an EvenT channel (FACET) by using AOP via AspectJ to decompose an implementation of the CORBA real-time event service into user-selectable features in order to enable code footprint minimization. The authors measured results by comparing code size among various configurations of FACET, in terms of both class file size and the size of natively compiled object files. They found a four-fold difference between a minimal configuration and one of the largest realistic configurations.

ACKNOWLEDGMENT

The authors thank Susan Anderson Klefstad for comments, revision, and help with analysis of empirical results.

REFERENCES

[1] Sun Microsystems, Java 2 Platform Enterprise Edition Specification Version 1.4, 2003, http://java.sun.com/j2ee/docs.html.

[2] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.6, 2001, http://www.omg.org/technology/documents/corba_spec_catalog.htm.

[3] Microsoft, .NET Framework, http://www.microsoft.com/net/.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, "Aspect-oriented programming," Proceedings of the European Conference on Object-Oriented Programming, vol. 1241, pp. 220-242, 1997.

[5] R. Klefstad, D. Schmidt, and C. O'Ryan, "Towards highly configurable real-time object request brokers," Proceedings of IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), 2002.

[6] P.A. Bernstein, "Middleware: A model for distributed system services," Communications of the ACM, vol. 39(2), pp. 87-99, February 1996.

[7] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Second Edition. Boston, MA: Addison-Wesley, 1999.

[8] Sun Microsystems, Connected Limited Device Configuration Specification Version 1.1, 2003, http://java.sun.com/j2me/docs/.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting Started with AspectJ," Communications of the ACM, vol. 44(10), pp. 59-65, October 2001.

[10] E. Lafortune, ProGuard, http://proguard.sourceforge.net/.

[11] C. Zhang and H.A. Jacobsen, Quantifying Aspects in Middleware, ACM AOSD, Boston, MA: 2003.

[12] F. Hunleth and R. Cytron, "Footprint and feature management using aspect-oriented programming techniques," Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems, pp. 38-45, 2002.

[13] The Real-Time for Java Expert Group, The Real-Time Specification for Java, http://www.rtj.org.