Resolving Feature Convolution in Middleware Systems

Charles Zhang and Hans-Arno Jacobsen
Department of Electrical and Computer Engineering
Department of Computer Science
University of Toronto
{czhang, jacobsen}@eecg.toronto.edu

ABSTRACT

Middleware provides simplicity and uniformity for the development of distributed applications. However, the modularity of the architecture of middleware is starting to disintegrate and to become complicated due to the interaction of too many orthogonal concerns imposed from a wide range of application requirements. This is not due to bad design but rather due to the limitations of the conventional architectural decomposition methodologies. We introduce the principles of horizontal decomposition (HD) which addresses this problem with a mixed-paradigm middleware architecture. HD provides guidance for the use of conventional decomposition methods to implement the core functionalities of middleware and the use of aspect orientation to address its orthogonal properties. Our evaluation of the horizontal decomposition principles focuses on refactoring major middleware functionalities into aspects in order to modularize and isolate them from the core architecture. New versions of the middleware platform can be created through combining the core and the flexible selection of middleware aspects such as IDL data types, the oneway invocation style, the dynamic messaging style, and additional character encoding schemes. As a result, the primary functionality of the middleware is supported with a much simpler architecture and enhanced performance. Moreover, customization and configuration of the middleware for a wide-range of requirements becomes possible.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Measurement

Keywords

Aspect Oriented Middleware, Middleware Architecture

1. INTRODUCTION

Middleware platforms, such as CORBA, DCOM, J2EE, and .NET, offer abstraction and simplicity for the complex and heterogeneous computing environment. They facilitate the development of high quality distributed applications with a shorter development cycle and a much smaller coding effort. Middleware systems are being adopted in a very broad spectrum of application domains, ranging from traditional enterprise platforms to mobile devices, embedded systems, real time systems, and safety critical systems.

General middleware design is particularly challenging because no assumptions can be made about the specific application domain of the middleware, neither should its architecture be coupled with any particular operating system or hardware platform. Consequently, *generality*, the designer's or vendor's interest to provide a set of commonly shared and reusable features, constantly wrestles with *specialty*, which represents the user's desire of having a tailored middleware to fit her specific needs. One solution to this dilemma is through multiple specifications and large product families. For example, within the common frame of CORBA technology, there is a proliferation of specifications such as CORBA, Realtime CORBA, Minimum CORBA, Data-parallel CORBA, and Fault-tolerant CORBA[1], and a proliferation of various product lines in which each member is engineered for specific domains including realtime environments, small memory devices, enterprise computing, and many others.[2] Newer middleware technologies, such as J2SE, J2EE, and J2ME, appear to have taken the same direction[3]. A serious limitation of these solutions, in addition to the increased complexity of development and maintenance, is that they only provide a fixed set of options for users, and, as a result, imperatively partition the application domains on behalf of the users. The mainstream of middleware implementations is "heavyweight, monolithic and inflexible" [9].

As many researchers have pointed out [9, 24, 25, 1, 21, 20], the effective solution to the problems described above

[1]These specification are collectively defined in [17] except for Data-parallel CORBA, which is located at `http://www.omg.org/technology/documents/specialized_corba.htm`
[2]The IONA product line includes Orbix Enterprise, Orbix standard, Orbix mainframe and ORBacus. `http://www.iona.ie/products/orbix.htm`. The OpenFusion product line includes embedded edition, realtime edition, and enterprise edition.
[3]Runtime libraries of Java provide a rich set of middleware functionality including RMI and CORBA.

is to achieve a high degree of configurability, adaptability and customizability in the middleware architecture. The ultimate goal is to customize middleware according to a specific user need, a concrete usage scenario, and a particular deployment or runtime instance. In many existing middleware implementations, this goal is unattainable because, as our past analysis shows [41, 43], many middleware features do not exist in modular forms and crosscut implementations of other functionalities. A majority of the current research efforts in alleviating the insufficiency of modularization aim at leveraging advanced programming models such as reflection [27] and component frameworks [31]. Component frameworks operate within the modularization capabilities of conventional programming languages, therefore, cannot effectively address concern crosscutting. Reflection, in addition to costly infrastructures, operates at a level of abstraction above programming languages, which often makes it difficult to write correct, easily readable and maintainable code. The aspect oriented programming paradigm, on the other hand, treats crosscutting concerns as first-class entities. Existing middleware applications of AOP [12, 16, 37] primarily focus on modularizing non-functional properties [13] as aspects and treat the middleware core as a monolithic building block. Our observations [41, 42, 43] reveal that the poor modularization of crosscutting concerns is an inherent phenomenon within this monolithic core. By "inherent" we mean that the failure of modularizing certain middleware features is not due to unwise design decisions but unavoidable using conventional programming paradigms. The implementations of these features do not have clear modular boundaries and are tangled among one another. We use the notion *"feature convolution"* to describe the deep entanglement of these functionalities in the middleware architecture.

As a remedy to this implementation convolution problem, we propose the method of horizontal decomposition (HD) and advocate the use of mixed-paradigms in middleware architectures. That is, we use conventional programming paradigms to provide *generality*: referring to a hierarchically decomposed architecture for a minimal, specialized, and commonly shared core; and we use aspect oriented programming to provide *specialty*: referring to domain-specific properties, which can be composed as aspects.

HD consists of a set of principles aiming towards the proper utilization of both paradigms in transforming the relationships among middleware functionalities from convoluted coupling to binary coupling. Our initial assessment of horizontal decomposition on a commercially deployed middleware product shows that the primary functionality of middleware can be supported with a much more coherent and simpler architecture with over 40% code reduction and around 8% performance improvement as compared to its original implementation. In addition, nine major functionalities ranging from various type supports to invocation styles become modular and selectable for customization. We are able to obtain over *sixty* [4] possible versions of the middleware; all through post-compilation transformations without changing a single line of source code.

This paper describes our approach and makes the following contributions:

[4]This rough calculation is based on 6 aspects and 2^6 possible combinations.

1. We present the *convoluted implementation problem* and describe the loss of modularity when decomposing multiple orthogonal design concerns using conventional decomposition methods.

2. We develop the method of *horizontal decomposition* as a set of principles for guiding aspect oriented decomposition of large middleware systems in order to address the convoluted implementation problem. Horizontal decomposition is a mixed-paradigm method exemplified on the analysis of a specific middleware implementation.

3. We evaluate the effectiveness of the horizontal decomposition principles and show that a number of innate and non-trivial middleware features can actually be implemented as aspects. This evaluation is performed through a case study of using aspect oriented refactoring on an industrial-strength CORBA implementation.

4. We provide a comparison of the refactored implementation to its original non-refactored counterpart through both structural metrics and runtime evaluation using a third-party performance benchmarking suite. This constitutes a quantitative evaluation of the effectiveness and benefits of the horizontal decomposition principles.

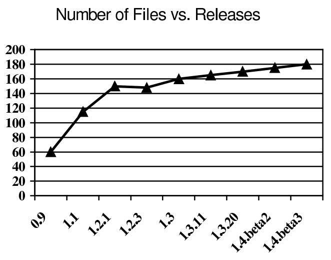
The rest of the paper is organized as follows. In the next section, we briefly define middleware, describe the recent architectural challenges, and introduce the adopted aspect language: AspectJ[5]. We then present the problem of feature convolution in Section 3. In Section 4 we develop the principles of horizontal decomposition and discuss these principles in light of middleware architecture design. Section 5 exercises the principles through the refactoring of a production strength CORBA implementation. A quantitative evaluation of the approach is presented in Section 6. A thorough discussion of related work is deferred to Section 7 to better position our work next to related approaches.

2. BACKGROUND

2.1 Middleware and Its Challenges

The term "middleware" has various interpretations. In this work, we focus on middleware that facilitates the development of distributed systems in a heterogeneous networked environment. Examples of these kinds of middleware implementations include CORBA, Java RMI, and .NET remoting. All of these are based on transparent remote procedure calls for providing a simplified network programming model. In recent years, in addition to traditional enterprise systems, middleware systems are being adopted in many emerging platforms such as routers[6], combat control systems [11], and wireless devices [6]. The new design requirements introduced by these platforms have catalyzed the fast evolution of middleware functionality and, in the mean time, created many problems for its architecture. Firstly, the structural complexity of middleware architecture increases

[5]AspectJ. URL: `http://www.eclipse.org/aspectj`
[6]Cisco ONS 15454 optical transport platform uses CORBA to deal with hardware customizations and communications between the management software and hardware drivers.



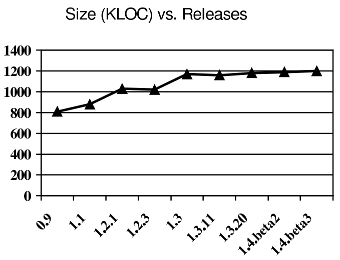


Figure 1: The Evolution of JacORB In Both Size and Modules.

dramatically. For example, as in Figure 1, the numbers of classes in JacOrb[7], an open source CORBA implementation in Java, increased around 50% in a development time of approximately four years. Its size has tripled during the same time. Secondly, in spite of the enriched functionality, the typical runtime of middleware platforms requires more and more computing resources, such as CPU, memory and power. For example, the minimum runtime of ORBacus Java[8], an industrial strength implementation of CORBA, requires around 10MB of memory. The C based CORBA implementation ORBit[9] requires around 2MB of memory space. This has become the main concern for using middleware systems on platforms with stringent resource constraints. For example, the typical memory space of newer wireless mobile devices is on the order of tens of mega-bytes with battery power of a few days[10]. In these cases, middleware systems are typically re-designed and separately implemented, and, at the same time, become too resource aware to satisfy enterprise computing needs.

2.2 Vertical Decomposition

We use the term "vertical decomposition" to denote the hierarchical decomposition of code modules advocated by many pioneers of software architecture including Dijkstra [10] and Parnas [28]. Hierarchical decomposition is based on levels of abstractions and step-wise refinements to divide-and-conquer complex problems. Hierarchical decomposition is mostly suitable for implementing a single independent function and performing a single logical task [3]. However, *"the drastic differences among aspects of complex systems are inherent and permanent, not mere artifacts of our current ways of doing things."* [39]. The rest of the paper uses the terms "veridical decomposition" and "hierarchical decomposition" interchangeably.

2.3 Aspect Oriented Programming

Aspect oriented programming offers a new paradigm for software development by complementing conventional programming paradigms with a higher degree of separation of concerns. Examples of aspects, often simply referred to as a system's "ilities" [13], include security, reliability, manageability, and, further, non-functional requirements. The existence of aspects is attributed to the use of the vertical decomposition paradigm to handle crosscutting concerns in software architecture. AOP overcomes this limitation by providing new language level constructs to modularize crosscutting concerns. The development of an aspect oriented application is commonly supported by a *component language*, such as Java or C, to implement the primary decomposition of a system; by an *aspect language*, such as AspectJ[11] and Hyper/J[12], to modularize crosscutting concerns as `aspects`; and also by the *aspect weaver* (a.k.a., aspect complier) that instruments the component program with aspect programs to produce the final system. AspectJ is one of the most mature aspect languages. In addition to conventional Java language features, AspectJ defines a set of new language constructs to modularize crosscutting concerns. For instance, a join point represents a well-defined point in the execution flow of the component program at which the AspectJ code can be executed. A `pointcut` construct denotes a collection of join points. For example, `cflow`, a *pointcut construct*, taking the definition of another pointcut, `Pointcut`, as its argument, "picks out each join point in the control flow of any join point P picked out by `Pointcut`, including P itself"[13]. AspectJ code can be executed *before*, *after* or *in place* of the program execution when a join point is reached. These actions are defined using AspectJ specific constructs `before`, `after`, and `around`. These constructs are called *advices*. An aspect module in AspectJ contains `pointcut`s and the associated `advice`s. It also contains *inter-type declarations*, which are used to declare new members (fields, methods, and constructors) in other types or to change Java type hierarchies.

3. THE IMPLEMENTATION CONVOLUTION PROBLEM

The implementation convolution problem refers to the phenomenon that, for a large number of non-trivial middleware functionalities, although their semantics are distinctive, their implementations do not have clear modular boundaries within the middleware code space and, more seriously, often tangle with one another. This prohibits these functionalities from being pluggable. Let us further exemplify this problem through CORBA [17] and its two common features, portable interceptors and the dynamic programming style[14]. Figure 2 illustrates the convolution among these two features as well as with the remote invocation mechanism, exemplified using classes from ORBacus. The class types are represented as vertical bars, of which the heights represent implementation sizes of these types. The figure shows that the implementation of the original client invocation mechanism (types `Downcall` and `DowncallStub`) becomes more complex with the additive code for addressing two other features, depicted as intermingled shades. Two additional class types are also created to implement the relationships among these three design concerns. Figure 3 shows implementation convolution in an actual ORBacus

[7]JacORB `http://www.jacorb.org`

[8]ORBacus: `http://www.orbacus.com`

[9]ORBit. URL:`http://www.gnome.org/projects/ORBit2/`

[10]Motorola A760 supports 32MB of memory `http://www.gsmarena.com/motorola_a760-392.php`

[11]AspectJ `http://www.eclipse.org/aspectj`

[12]Hyper/J `http://www.alphaworks.ibm.com/tech/hyperj`

[13]The AspectJ Programming Guide. URL:`http://www.eclipse.org/aspectj/`.

[14]Interceptors are standardized callback mechanism in CORBA to allow the registration of additional CORBA services. The dynamic programming style refers to the reflective invocation mechanism of CORBA.

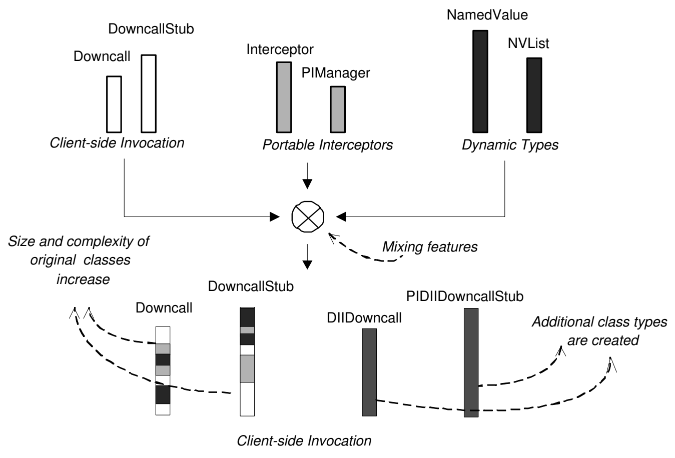
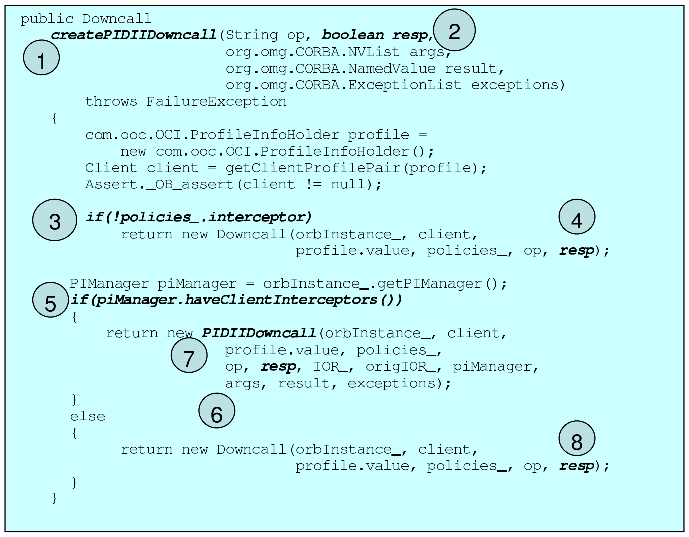


Figure 2: Implementation Convolution in ORBacus.



```
public Downcall
   createPIDIIDowncall(String op, boolean resp,
                       org.omg.CORBA.NVList args,
                       org.omg.CORBA.NamedValue result,
                       org.omg.CORBA.ExceptionList exceptions)
      throws FailureException
   {
      com.ooc.OCI.ProfileInfoHolder profile =
         new com.ooc.OCI.ProfileInfoHolder();
      Client client = getClientProfilePair(profile);
      Assert._OB_assert(client != null);

      if(!policies_.interceptor)
         return new Downcall(orbInstance_, client,
                             profile.value, policies_, op, resp);

     PIManager piManager = orbInstance_.getPIManager();
     if(piManager.haveClientInterceptors())
     {
        return new PIDIIDowncall(orbInstance_, client,
                    profile.value, policies_,
                    op, resp, IOR_, origIOR_, piManager,
                    args, result, exceptions);
     }
     else
     {
         return new Downcall(orbInstance_, client,
                             profile.value, policies_, op, resp);
     }
   }
```

Figure 3: Convoluted Code: Interceptor, DII and oneway, all tangled together.

code snippet. In this short piece of code, three concerns are present: places 3 and 5 deal with interceptors, 2, 4, 6, and 8 with `oneway` calls which will be discussed in detail in later sections, and 1, 7 with support for both interceptors and the dynamic programming style. This type of problem is not due to the design limitations of ORBacus. Our examination of three different CORBA implementations [41, 43] shows that around 50% of the classes coordinate with a second design concern. Moreover, 10% of these classes coordinates with three and more concerns. The phenomenon of *crosscutting* arises "whenever two properties being programmed must compose differently and yet be coordinated" [23]. We extend this AOP term and use *implementation convolution* to describe this large scale N-by-N crosscutting phenomenon.

Implementation convolution means the loss of modularity and configurability. It also incurs runtime overhead since, in a particular middleware deployment or runtime instance, not all functionalities are required to participate in the main operational logic of the middleware. However, these functionalities still exist in forms of class variables, method arguments, and branching conditions, which constitute parts of the overall execution path, the application memory space, and the program control flow. For example, although the asynchronous "oneway" invocation semantics is not used in many CORBA applications, it is not yet possible to flexibly load or unload this feature in today's middleware architectures due to, as illustrated above, its non-modular implementation. To enable customizability of this feature, its implementation must be separately modularized outside of the middleware core. But before proceeding to this action, it is necessary to answer two fundamental questions:

1. Why should a particular functionality, such as *oneway*, be treated as an aspect?
2. What steps are required to untangle convoluted features?

The next sections are devoted to answers of these questions.

4. HORIZONTAL DECOMPOSITION

4.1 Overview of Horizontal Decomposition

The goal of horizontal decomposition (HD) is to achieve a mixed-paradigm architecture in which the conventional decomposition methods and the aspect oriented approaches are used together, each addressing a different category of functionalities with its maximum strength. Horizontal decomposition comprises a set of guidelines to, firstly and most importantly, distinguish aspect functionalities from non-aspect ones in order to lay out clear responsibilities for AOP and, secondly, to enable super-impositional architectures. By "super-imposition" we mean that, in the context of horizontal decomposition, implementations of aspects can be transparently applied onto a generic core through the "weaving" process to achieve the desired customization of middleware functionality. We use the term "horizontal" to emphasize its complementarity and its synergistic co-existence with the "vertical" decomposition. In the rest of this section, we first abstractly present and discuss the HD principles. We then discuss the principles' application within the context of middleware architecture. In Section 5, we implement and validate the principles through a refactoring based aspect oriented middleware architecture approach.

4.2 Horizontal Decomposition Principles

The horizontal decomposition method consists of five principles synthesized from our past experience [41, 42, 43] and the ongoing application of AOP to middleware architecture. These principles are listed following a logical order in which they can be sequentially applied.

Principle 1: Recognize the relativity of aspects. *The semantics of an aspect is determined by the primary functionality of the application.*
From the definition of concern crosscutting, the semantics of an aspect can only be determined with respect to the primary function of the application. For example, logging, a well known crosscutting concern, does not crosscut the logging facilities itself. A further example draws from mid-

dleware implementations that specializes in making remote invocations; there, the efficient invocation of local servers is recognized as a crosscutting concern (cf. Section 4.3). However, in the context of non-distributed applications, the remote invocation mechanism can be implemented as an aspect [35]. We use these examples to highlight the possible ambiguity for the semantics of aspects in large and complex systems. This ambiguity should be clarified as much as possible because we believe aspects and non-aspects ought to be handled in different ways. The recognition of the relativity property is the first step of applying aspect oriented decomposition.

Principle 2: Establish the coherent core decomposition. *The basis of aspect oriented decomposition is the establishment of a functionally coherent and vertically decomposed core.*
Cohesion, first introduced in [38], expresses the degree of association between components in a module. Among the multiple levels of cohesion, the most desired level is *functional cohesion* where "every function within a module contributes to directly performing a single task" [3]. The semantics of aspects has to be discussed with respect to an architecture which, ideally, should be functionally coherent and does not contain convoluted features. We refer to this referential architecture as the core decomposition, or just simply "core". In large software systems, such as middleware, the core consists of several conceptual components [36][15]. Each of the components focuses on a single task and they are logically coherent in supporting the primary system functionality or its most typical usage. For this reason, it is minimal and simplistic. For example, since the primary functionality of a telecommunication system is call processing, its core is the basic call processing system excluding features such as call waiting or call forwarding [40]. Our definition of a core has the following two benefits: 1. It is easier to obtain efficient hierarchical decompositions if the system only supports a limited number of functionalities. 2. Since the core captures the most essential functionality of a software system, we can use it as the basis for further customization.

Principle 3: Define the semantics of an aspect according to the core decomposition. *Using the core as a reference, a functionality is considered orthogonal if both its semantics and its implementations are not local to a single component of the core. Only the orthogonal functionality is treated in the aspect oriented way.*
Our definition of aspects is more aggressive and more precise as compared to ilities [13], quality-of-service (QoS) requirements [12], or general distributed computing concerns [4]. For instance, the customization of communication protocols could be described as both an ility (customizability) and a distributed concern, and, hence, could be classified as an aspect. However, it is not an aspect by our definition because its semantics are likely local to the communication component of the application. In practice, techniques such as component frameworks can confine this customization within the protocol layer of the architecture as in the OCI plug-in framework used by ORBacus and the ACE framework[16]. On the other hand, the asynchronous invocation style of middleware (i.e., oneway semantic), however, is an aspect because it requires not only the non-blocking support from communication protocols and additional request processesing routes but also the corresponding programming model in the service description language. Studies on AOP implementation of design patterns [18] show that certain concerns are better addressed by conventional techniques than by AOP, and vice versa. It is then crucial to avoid ambiguity of the semantics of aspects as much as possible in order to maximize the modularization capabilities of both vertical decomposition and the AOP paradigm.

Principle 4: Maintain a class-directional architecture. *Crosscutting concerns should be implemented class-directional towards the core.*
Class-directional is a category of relationships between base modules (classes) and aspects in which aspects know about the class but not vice-versa" [22]. Class-directional in HD means the system core does not have the knowledge of aspect implementations. Our previous work shows that middleware aspects, such as the interception support and the dynamic invocation semantics, can be completely separated from the core implementation and transparently super-imposed back [41, 43]. Later in this paper, we show that maintaining class-directional can even be achieved for crosscutting concerns of a much larger scale. The property of class-directional does imply a strong dependency of aspect implementations on a fairly stable architecture of the core. This is because, if the model of the core architecture evolves too quickly over time, the semantics of an aspect has to be modified correspondingly due to the relativity principle. However, we believe a stable core architecture is a natural outcome of the horizontal decomposition. With a single or a few design goals, the architecture tends to be focused and stable. Design patterns [15] are excellent examples of stable architectural ideas being repeated many times for specialized problems.

Principle 5: Apply incremental refactoring. *Decomposition in the aspect dimension is assisted by incremental refactoring.*
The establishment of the coherent core often requires a series of refinements. There are at least two reasons for this to happen. First, the identification of the core in complex systems is not always straightforward and could be completed gradually. Second, the composition of the core can be viewed at different levels of granularity. In other words, a functionality well localized within a single component can become scattered, hence, crosscutting, if this component is further factored into several parts. The refinements of the core semantics can result in the discovery of new aspects, and refactoring must be performed to resolve their convolution with the newly established core as well as with the code of existing aspects. For example, during our resolution of the convolution presented previously in Figure 3, it had not occurred to us that oneway is an aspect until after DII and PI were already refactored. Two types of refactoring were then performed: 1. Re-factorization of oneway out of the core; 2. Re-factorization of oneway out of aspect DII and aspect Interceptor Support. We refer to these two types as the first and second degree refactoring. The horizontal decomposition is, hence, conducted in an incremental and accumulative fashion. We illustrate this process further in Section 5.3.2.

[15]In this paper, we use the term "component" in short for conceptual component. A conceptual component can be mapped to one or more physical components.

[16]The Adaptive Communication Framework. URL: `http://www.cs.wustl.edu/~schmidt/ACE.html`

4.3 Application to Middleware Architecture

In this section, we further explain the horizontal decomposition principles through a discussion of their application to the aspect oriented analysis of the middleware architecture. This analysis is carried out in the following logical steps:

Middleware aspects are relative to the primary functionality of middleware. (Principle 1) The relativity principle prescribes that the semantics of an aspect must be discussed within the context of the application, i.e., in our case, the middleware architecture itself. Therefore, we oriented ourselves, prior to the detailed analysis, as follows: we first define the primary functionality of middleware as the support for transparent remote invocation; we then define a middleware aspect as a middleware feature that crosscuts the implementation of this functionality.

The middleware core consists of a set of components that support transparent remote invocations. (Principle 2) To establish the basis for the semantics of middleware aspects, we firstly define the "middleware core" as the mechanism of *composing, transporting*, and *dispatching* invocation requests in enabling transparent remote invocations. This mechanism is supported by service description languages and the associated stubs/skeletons, service identification mechanism, request dispatching mechanisms, and transport protocols. Table 1[17] gives concrete examples of these components in popular middleware implementations. Since remote invocations are emulated as normal method calls, the middleware core also needs to support various data types in the description languages, synchronous/asynchronous communications, and statically or dynamically typed requests. We aggressively simplify the middleware core to only support one primitive type, the synchronous invocation style, and the statically typed requests. All other functionalities are treated as customization options.

Middleware aspects are features that cannot be encapsulated within an individual component of the core decomposition (Principle 3). Let us further exemplify this concept using the CORBA feature of "server collocation" as an example. One of the drawbacks of transparent remote invocation is that the location of the remote service is hidden and could be in the same process as the client. A common optimization is that middleware transparently detects this situation and directly dispatches the request without going through the network layer. In the ORBacus implementation of CORBA, a normal remote invocation traverses the middleware stack in the following logical order (Figure 4): client marshalling of data, the transport of data in the protocol layer, server dispatch of the request, and server unmarshalling. Figure 5 illustrates the invocation sequence when this optimization is added. Not surprisingly, the sequence becomes more complex. This optimization creates a "logic glitch" because the client request traverses into the "dispatch layer", a server side component (Figure 4), inside the "Protocol" layer at the client-side. The hallow arrows represent program logic corresponding to the optimization, and the shaded boxes represent the activities spent in performing the optimization. It is not hard to conclude that the *Local Invocation Optimization* is an aspect because it

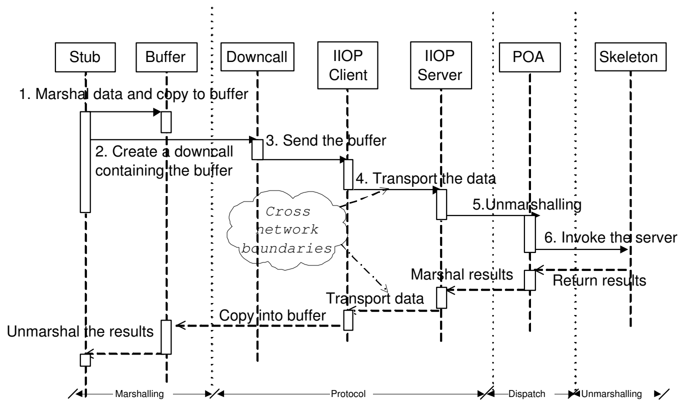


Figure 4: Remote Invocation: ORBacus

requires the collaboration among three components: Marshalling, Protocol, and Dispatch. It is noteworthy that this optimization only addresses in-process servers. Adding further optimizations for in-host servers undoubtedly complicates the picture even further. Two main disadvantages for this implementation can be completely overcome if modularized in aspects instead: 1. The implementation scatters around different parts of the core architecture which makes it hard to understand and change. 2. The inability of plugging out this feature incurs redundancy in the execution of remote method calls.

The implementation of middleware aspects are class-directional and super-impositional. (Principle 4) We characterize super-impositional middleware architecture as follows[18]:

1. Multiple sets of vertical decompositions: As will be described in Section 5.3.1, we distinguish the functionality of an aspect from its crosscutting interaction with the core. The core of the application and the functionality of aspects are separately decomposed into vertical hierarchies of modules. Each can be compiled independently. The core decomposition is fully operational in supporting the primary functionality of the application.

2. Exclusive application of AOP to the crosscutting logic of an aspect: The crosscutting logic is the interaction between the functionality of an aspect with both the core and, if necessary, other aspects. This interaction is the only place where AOP is applied, and we metaphorically refer to the architecture of the interaction as the "glue" architecture.

3. Flexible combination of architectures: The goal of the super-impositional architecture is that the combination of the middleware core and aspect functionality is flexible and conducted at the post compilation stage, e.g., through source code transformation as in AspectC++[19], or bytecode weaving as in AspectJ, or

[17]In this table, ROT refers to Running Object Table. `http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.asp`.

[18]By "super-imposition" we mean that, in the context of horizontal decomposition, implementations of aspects can be transparently applied onto a generic core through the "weaving" process to achieve the desired customization of middleware functionality.

[19]AspectC++ `http://www.aspectc.org`

	CORBA	*DCOM*	*.NET Web Services*
Description Language	IDL	MIDL	C#,CLR languages
Identity Publication	IOR	OBJREF	WSDL File
Request Dispatching	POA	ROT	ASP.NET process
Protocol	IIOP	Object RPC	SOAP

Table 1: Core Middleware Architecture Elements.

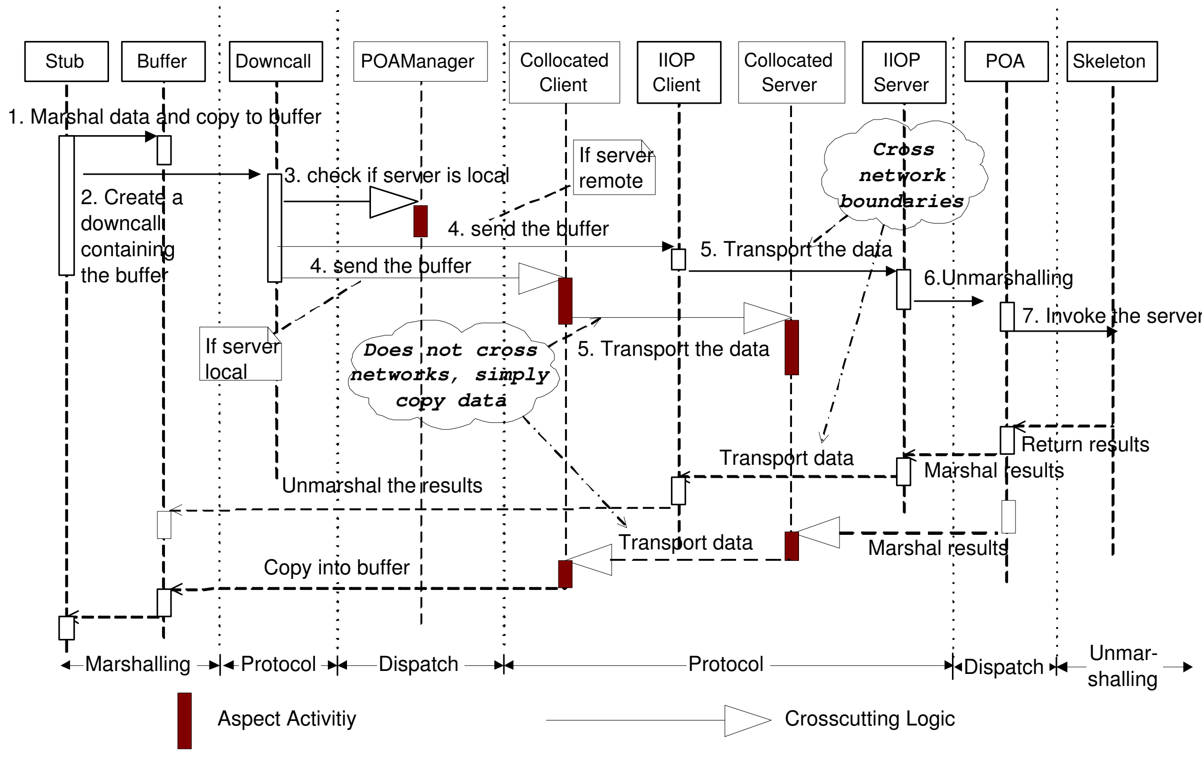


Figure 5: Addressing Remote and Local Invocations Simultaneously in ORBacus.

even runtime weaving [44]. Therefore, the "glue" architecture of an aspect must address its interactions with the core and other aspects individually. This is a desired property because it reduces the complexity of the implementation from a convoluted relationship to a set of binary relationships.

We show in later sections that this super-impositional architecture for middleware can be achieved. We do have to modify how certain core semantics are expressed in the code in order to obtain the necessary contexts in AspectJ pointcuts. However, none of the modifications change our core decomposition model, and, secondly, this might be a language-specific phenomenon of AspectJ.

Middleware aspects are implemented incrementally. (Principle 5) Refinements of the definition for the core decomposition give rise to the identification of new aspects. To alleviate this problem, we use both the first and the second degree refactoring to separate new aspects from both the middleware core and previously identified aspects. Our experience shows that the second degree refactoring, contrary to intuition, does not cause major changes to existing aspect code. This is because many aspects identified and implemented at early stages mainly crosscut the core within the body of procedures. We refer to these aspects, such as logging, tracing, and certain type manipulations, as code-level aspects in contrast to the architectural aspects, which crosscut the middleware core at the level of attributes and methods. The first degree refactoring of these aspects, such as dynamic programming styles, typically involves pruning core objects at the method level, while preserving most of the intra-procedural interactions or pointcuts of the code-level aspects.

5. RE-FACTORING BASED IMPLEMENTATION OF HORIZONTAL DECOMPOSITION

We choose to use the technique of refactoring [14] to evaluate the horizontal decomposition principles. We use refactoring since it conveniently allows us to focus on modular compositions through the re-use of design decisions and to systematically and fairly compare horizontally decomposed middleware with its conventional counterpart. With no loss of generality, we use CORBA, one of the most developed middleware technologies, as a case study, and ORBacus, an industrial strength Java CORBA implementation, as the target of our re-implementation. The AOP language we choose is AspectJ. We have identified and refactored a total of nine major functionalities as aspects in ORBacus. Most of them can also be found in other CORBA or even non-CORBA middleware systems. Our implementation shows that horizontal decomposition can deliver its promise. We have obtained a much more concise and efficient middleware core which, at the time of writing this paper, exhibits around 8% improvement in benchmark performance and over 40% reduction in code size, with ample room for further improvements. In addition, we have a super-impositional architecture in which combinations of these nine aspects can be freely selected to form new versions of ORBacus supporting both the core functionality and the aspectual functionality. In the following sections, we describe our implementation in detail and present the evaluation results.

5.1 Defining the Middleware Core

Our reference model of the ORBacus core consists of the following layers listed top-down. in accordance with Table 1. Each layer performs one specific task. Identity publication is a specialized CORBA operation. The corresponding implementation in ORBacus is compact and coherent and, hence, omitted from the list.

1. IDL Layer: Stub and Skeleton. The function of stubs and skeletons, generated from a service description language, is to support the masking of remote invocations as local method calls at the interface level. They are common middleware elements serving as translators between application semantics and the middleware substrate. Our working definition of minimum stubs and skeletons only supports statically typed invocations, the synchronous invocation style, and the IDL definitions for essential primitive data types such as integer. In other words, the descriptions of advanced features are not enabled by default. These features include: advanced data types such as `Any`, multi-byte characters, invocations through reflection such as DII or DSI, the asynchronous invocation style denoted by the `oneway` keyword in CORBA's IDL. These features are "woven" into the stubs and skeletons by an aspect-aware IDL compiler [42].

2. Messaging Layer: Client-side and Server-side. This layer consists of two conceptual components: the client-side "downcall" mechanism responsible for marshalling the requests and the server-side "upcall" mechanism responsible for unmarshalling and request dispatching. Corresponding to our definition of the minimum skeleton and stub, the "downcall" and "upcall" mechanisms should only support primitive data types, synchronous and statically typed invocations.

3. Transport and Protocol Layer. This layer handles the communication with peer ORBs using IIOP. ORBacus implements the Open Connector Interface (OCI) (i.e., pluggable transports) based on acceptors and connectors [30]. We define this layer to only support the synchronous communication and no interoperability with different code sets (i.e., character encoding schemes.)

We do not claim that this core model is crosscutting free. Each component can be further broken down into finer logical constituents. Nevertheless, it is coherent enough for us to apply AOP to a large number of ORBacus features.

5.2 Defining CORBA Aspects

As previously stated, a middleware functionality can be classified as an aspect if it interacts with multiple components. Below, with omission of the internal details of CORBA, we summairze the logical independence (orthogonality) of five aspects, their functional intend (semantics), and characterize their original crosscutting implementation.

I ***Oneway invocation semantic.***
Semantics: Supports the best-effort and asynchronous delivery of client requests. No response is expected.
Orthogonality: The core supports the synchronous invocation semantic.
Crosscutting: *IDL Layer*: The support of IDL keyword "oneway". *Messaging Layer*: Additional logic in the "downcall" process for not expecting a response as well as in the "upcall" process for no need to issue a response. *Protocol Layer*: The support of GIOP encoding of the oneway flag as well as the setting of a timeout value for the socket.

II ***Dynamic typing.***
Semantics: Supports reflective composition of remote invocations. Any and Dynamic Any (DynAny) are used to represent arbitrary IDL data types including primitive types and abstract ones. `Typecode` is used to encode the type information
Orthogonality: The core supports statically typed invocation requests.
Crosscutting: *IDL Layer*: The support of dynamic IDL data types such `Any`, `Dynamic Any` and the associated stub/skeleton operations. *Messaging Layer:* The marshaling and unmarshalling of these data types. *Protocol Layer:* None. Data is treated as byte streams.

III ***The wchar and wstring support.***
Semantics: Supports the expanded character sets such as Unicode[20].
Orthogonality: We view IDL data types as independent, hence, orthogonal ways of encapsulating and interpreting the transported bytes.
Crosscutting: *IDL Layer:* The support of `wchar` and `wstring` IDL data types and the associated stub/skeleton operations. *Messaging Layer:* The marshalling and unmarshalling of these data types. *Protocol Layer:* None. This layer treats all data as byte streams.

IV ***The encoding conversion.***
Semantics: Supports transparent conversions for the data exchange, as part of the interoperability support of CORBA, if the communicating ORBs use different character encoding schemes.
Orthogonality: The functionality of managing different character encoding schemes is clearly logically independent of the semantic of the CORBA core which manages transparent remote invocations.

[20]Unicode. URL: `http://www.unicode.org`

Crosscutting: *IDL Layer:* None. The functionality is transparent to applications. *Messaging Layer:* Adding logic to both "downcall" and "upcall" processes as to decide if conversion should take place when reading and writing characters to the data buffer. *Protocol Layer:* Adding logic to the server side protocol layer which builds the conversion schemes for an incoming request based on the encoding information in GIOP, before passing it up to the messaging layer.

V ***The local invocation support.***
Semantics: Supports direct forwarding of requests if the remote service is located in the same process.
Orthogonality: Local invocation is logically orthogonal to remote invocation functionality of the CORBA core.
Crosscutting: Please refer to Figure 5 for details.

Concluding from the analysis presented above, these crosscutting features have both design and runtime implications. Their implementation is scattered and, thus, "hidden". It is hard for programmers to change and to maintain them. Though often optional in normal operations of CORBA, these features are always initialized and evaluated during the execution. This runtime redundancy degrades the performance of the core as confirmed by our evaluation. Recent dynamic compilation techniques can provide solutions to eliminate runtime redundancy. However, more coherent application semantics are always more effective in performance improvements.

5.3 Resolving Implementation Convolution

The goal of our aspect oriented treatment is to eliminate the convoluted features in the original code base through modularizing orthogonal functionality as aspects and untangling of the code convolution among aspects themselves. The next two sections provide detailed descriptions of these two stages.

5.3.1 Implementing Middleware Aspects

Our implementation of middleware aspects generally consists of two distinct parts: the implementation of the aspect functionality itself, which is best handled in a hierarchical decomposition; and the implementation of the interaction between this aspect and the core, which is decomposed in the aspect oriented way. For example, the complete implementation of the support for codeset conversion, as illustrated in Figure 6, consists of the implementation of its functionality (left) and of its crosscutting logic with the core (right). Its functionality is decomposed into a normal Java class hierarchy which embodies the basic design rational of composing a converter from both a "Reader" and a "Writer". The hierarchy of the crosscutting logic consists of three aspects representing three different parts of the crosscutting logic including the conversion of character streams, the setup of conversion utilities, and the error handling regarding conversions. Figure 7 shows a specific implementation instance in supporting conversion of character streams. The area enclosed by the dotted box in the original implementation (left) represents the crosscutting logic and it is re-implemented as an "around" advice (lower-right). This advice, when "woven" into the core implementation by the AspectJ compiler,

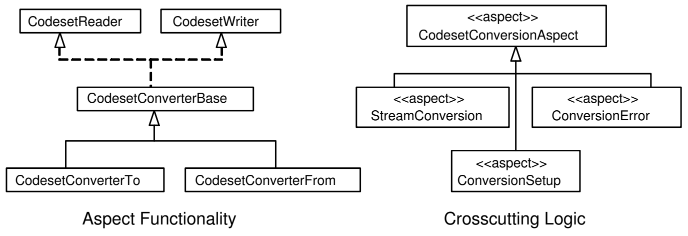


Figure 6: Aspect Implementation of Character Conversion.

replaces calls to the method `InputStream.read_char` as follows: it `proceed`s to the normal `read_char` call (upper-right) if conversion is not necessary; otherwise it creates a converter and performs conversion before returning. With this we have achieved a dramatic simplification of the original core implementation, `read_char`, as well as the preservation of its functionality together with the aspect code.

The separation of functionality and crosscutting logic, combined with refactoring, benefits us in the following ways:

1. The original design choice is fully respected and preserved. There is no shifting of programming paradigms in implementing the functionality of aspects. Hence, the domain expertise embedded in the design is left intact. Even for new implementations, our approach places no restrictions on the use of the vast and rich repertoire of vertical design techniques such as design patterns.

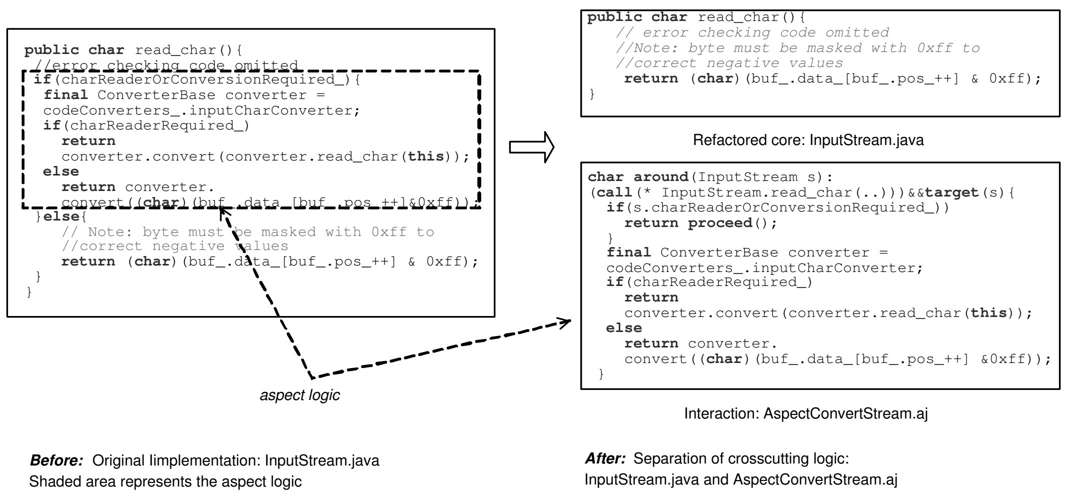
2. The crosscutting logic is isolated and, therefore, can be conveniently analyzed for the discovery of implementation patterns. By patterns we mean the concerns commonly addressed while implementing the crosscutting logic. We describe some of the patterns we observed from our initial implementation later in this section.

3. The separation of the aspect functionality and its crosscutting logic is explicit and can be completely decoupled. Similarly to the advantage of separating an interface and its implementation in the objected oriented paradigm, we believe this separation is fundamental in supporting the plug-and-play of new aspects such as a new invocation style or a new character encoding scheme. However, a thorough exploration of this property is outside the scope of this paper.

We feel that a good decomposition of the crosscutting logic is the most challenging task in horizontal decomposition. Deferring a more serious analysis and formulation to our future research, we summarize our experience as observations of commonly addressed crosscutting logic patterns in our implementation of aspects:

1. Lifecyle. Lifecyle crosscutting intercepts the set-up and the tear-down stages of the core and performs the initialization and the destruction of aspect-specific utilities. This crosscutting pattern is present in all aspects except "oneway" which does not have the functionality implementation. Because of this temporal relationship, this type of crosscutting logic is commonly implemented in `before` and `after` advices.

2. Data. Data crosscutting enables different views of the same data stream by adding APIs to different components of



```
public char read_char(){
 //error checking code omitted
 if(charReaderOrConversionRequired_){
  final ConverterBase converter =
  codeConverters_.inputCharConverter;
  if(charReaderRequired_)
    return
    converter.convert(converter.read_char(this));
  else
    return converter.
    convert((char)(buf_.data_[buf_.pos_++]&0xff));
 }else{
    // Note: byte must be masked with 0xff to
    //correct negative values
    return (char)(buf_.data_[buf_.pos_++] & 0xff);
 }
}
```

```
public char read_char(){
   // error checking code omitted
   //Note: byte must be masked with 0xff to
   //correct negative values
    return (char)(buf_.data_[buf_.pos_++] & 0xff);
}
```

Refactored core: InputStream.java

```
char around(InputStream s):
(call(* InputStream.read_char(..)))&&target(s){
  if(s.charReaderOrConversionRequired_))
    return proceed();
  }
  final ConverterBase converter =
  codeConverters_.inputCharConverter;
  if(charReaderRequired_)
    return
    converter.convert(converter.read_char(this));
  else
    return converter.
    convert((char)(buf_.data_[buf_.pos_++] &0xff));
 }
```

Interaction: AspectConvertStream.aj

Before: Original Iimplementation: InputStream.java
Shaded area represents the aspect logic

After: Separation of crosscutting logic:
InputStream.java and AspectConvertStream.aj

Figure 7: Implementation of Crosscutting Logic: Code Example.

the core. These APIs support the instantiation of new data types, conversions between streams and data types, and conversions between different data types. Aspects that heavily exhibit this type of logic crosscutting include dynamic types, the wide character support, and codeset conversion support. Data crosscutting is mostly implemented as methods which are "woven" into the class definition of the core classes via the inter-type declaration mechanism of AspectJ.

3. Error. Error crosscutting augments the error handling mechanism of the core with that of the aspect. It primarily involves adding aspect-specific error codes and descriptions using inter-type declarations. It also involves the validation of the states of aspects prior or posterior to core method calls. These checks are naturally captured in `before` and `after` advices of AspectJ. This crosscutting logic can be found in all the aspects.

4. Messaging. Messaging crosscutting alters the normal invocation sequence of the core to support different invocation styles. This type of crosscutting logic is usually found in aspects that add alternative invocation paths to the core components, and, hence, does not have any aspect-specific functionality implementation. The implementation of aspects, such as "oneway", extensively uses the control-flow join points of AspectJ, such as `cflow` and `cflowbelow`.

5.3.2 Untangling Convoluted Aspects

As previously described, aspects can also crosscut each other in supporting a complex functionality in middleware implementations. We have depicted such a scenario in Figure 3 in which three aspects, interceptor support, the dynamic requesting style, and oneway, are convoluted in providing interceptable-dynamic-oneway requests. Let us use this same example to illustrate how convoluted implementations can be resolved. Figure 8 shows simplified code snippets of our resolution. The aspect code snippet 1 only deals with DII by adding a method `createDIIDowncall` to the core class `DowncallStub` (line 2) for the creation of a DII downcall using an inter-type declaration. Code snippet 2 uses AspectJ's capability of return-value modification, `after returning` (line 1), and changes the return value of `createDIIDowncall` to its subtype `PIDIIDowncall` (line 7) if interception is enabled (line 3). Code snippet 3 uses the same return-value modification feature to set the `response-Expected_` flag of the `Downcall` to either true or false depending on whether the request is marked "oneway" or not in the global hashtable (line 8). We have improved over the original implementation with better cohesion as each code snippet in Figure 8 is specialized in providing one particular functionality. We have also untangled a convoluted relationship into a set of simpler binary relationships: DII-core, PI-DII, and Oneway-DII. Through the use of the byte-code weaver, we can configure the following seven versions of ORBacus without touching the source code: Plain ORBacus (CORE), CORE+PI (Portable interceptor), CORE+-DII, CORE+Oneway, CORE+PI+DII, CORE+PI+Oneway, CORE+DII+Oneway.

We use the matrix in Table 2 to summarize the convoluted relationships among the aspects that we have identified so far. Each "x" in the table means the row aspect crosscuts the column aspect. We also include three aspects identified in our previous work [41, 43], including portable interceptor support (PI), local invocation (LI), and the dynamic programming interface (DPI), as these relationships were not explored previously. We purposely leave out the core architecture since every aspect crosscuts the core by default.

	LI	Conv	Dyn	DPI	PI	OW	Wchar
Conv.	x	N/A	x		x	x	x
Dyn	x		N/A	x	x		x
PI				x	N/A		
OW	x			x		N/A	
Wchar		x	x		x		N/A

Table 2: Convolution Matrix. (Conv: Conversion. Dyn: Dynamic Typing. OW: Oneway. CO: Collocated Server. DPI: Dynamic Programming Interface. PI: Portable Interceptor. Wchar: Wide character and wide string).

```
public Downcall
DowncallStub.createDIIDowncall(String op,//arguments
omitted)
throws FailureException {
ProfileInfoHolder profile =new ProfileInfoHolder();
   Client client =
   getClientProfilePair(profile);
   Assert._OB_assert(client != null);
   return new Downcall(orbInstance_, client,
    profile.value, policies_, op);
}
```

(1) Adding DII to core

```
after(DowncallStub s): returning (Downcall downcall)
&& target(s)&&call(createDIIDowncall(..)){
 if(!s.policies_.interceptor)
   return downcall;

 PIManager piManager=orbInstance_.getPIManager();
 if(piManager.haveClientInterceptors()){
   return new PIDIIDowncall(//arguments omitted);
 }
 else{
    return downcall;
  }
}
```

(2) Adding PI to DII

```
aspect oneway {
Hashtable responseflgs = new Hashtable();
//response table is initialized as an earlier stage
after(DowncallStub s) returning (Downcall downcall)
&&target(s)
&&call(* DowncallStub.createDIIDowncall(..){
 Object flag = responseflgs.get(s);
 downcall.responseExpected_= (flag==null);
}
```

(3) Adding oneway to DII

Figure 8: Resolving convolution of aspects: Code Example.

5.4 Incremental Decomposition: A Retrospective

It is hard to identify all aspects due to the difficulty of defining the convolution-free core of the system. This is partly because of the limitations of existing aspect discovery techniques and tools. It is also because the definition of the core architecture is not likely to be very precise at the beginning of the decomposition process. Our experience shows that the definition of the minimal core is adjusted and refined gradually over time. Consequently, we continuously discover new aspects as our definition of the core architecture evolves. The complete untangling of new aspects involves both their separation from the core and previously identified aspects. Therefore, the complete aspect decomposition model is obtained in an incremental fashion since each identification of new aspects possibly triggers both first and second degree refactoring. Table 3 summarizes our decomposition process of the aforementioned aspects retrospectively, where we list our implementation stages of aspects in a chronological order. For instance, our initial refactoring (Stage 1) starts with aspects Portable Interceptors (PI), Dynamic Programming Interface (DPI), and Collocated Server (CO) listed in Column A while the other aspects (Column B) are yet to be identified. The subsequent refactoring (Stage 2) of the oneway (OW) aspect involves

Stage	A	B	C
1	PI, CO DPI	Dyn, Wchar, Conv, OW	
2	OW	Dyn, Wchar, Conv	DPI, PI CO
3	Dyn	Conv,Wchar	CO, DPI, PI
4	Wch	Conv	Dyn, PI
5	Conv		CO, OW, PI, Dyn, Wchar

Table 3: Incremental Decomposition of Aspects (A: Aspects being refactored. B: Aspects contained in core. C: Aspects being refactored in 2nd phase. Abbreviations are the same as in Table 2.

modifying not only the core but also the three aspects in Column C (Row 2). This table shows that both the first and the second-degree refactoring play important roles in resolving the convolution.

As the result of keeping no knowledge of aspects in the core, the architecture of the refactored ORBacus is self-contained. The basic functionality of CORBA is preserved. In fact, our CORBA core coincidently fully supports all operations of a third-party benchmarking tool. Meanwhile, combinations of the horizontal features can be selected and transparently configured into the core architecture through different build files and the AspectJ compiler [21]. There is no restrictions to the combinations except portable interceptors, which requires the support of type `Any`.

6. IMPLEMENTATION EVALUATION

The emphasis of our evaluation is to measure how well the principles of horizontal decomposition deliver their most important promise — supporting the core functionality of middleware more efficiently in a much less convoluted architecture. We divide the evaluation for our refactoring-based implementation into two parts. We first measure, using standard metrics, the architectural changes over the original implementation as the result of decomposing a number of major middleware features in aspect modules. We are interested in, while supporting the same core functionality, how much more concise middleware architecture has become with implementation convolution resolved. We then perform the performance evaluation by comparing the horizontally decomposed ORBacus core with the original implementation in supporting a set of standard CORBA functionality provided using a third-party benchmarking suite.

6.1 Structural Comparison

To measure the structural differences, we employ a set of standard software engineering metrics which we refer to as structural metrics. They are explained in great detail in [41]. These metrics include the following: size of the executable source code, cyclomatic complexity, weight of the class and efferent coupling. We first measure the direct impact of horizontal decomposition on the entire ORBacus implementation, i.e., all classes in the `com` package hierarchy. This includes both the functionality of aspects and their interaction logic with the core. Table 4 shows that, by applying horizontal decomposition and stripping out crosscutting functionalities, we have reduced the size of the ORBacus core by

[21]The AspectJ compiler simply issues a warning for applying aspects to un-found classes.

around 10K lines or 42% of code, 855 or 35.6% fewer methods, around 17% simplification in terms of the control flow, and 22% reduction of coupling. This shows that, in spite of its rich set of functionality, the original implementation is monolithic and "oversized" for common CORBA operations. Our refactored version is much lighter and much more flexible for configuration and customization.

Table 5 presents a different perspective for the resolution of the crosscutting logic in the ORBacus core classes. We count the reduction of three types of language elements that have further runtime implications. The reduction of arguments for methods and constructors not only allow the semantics of classes to be expressed more concisely but also enables more energy-efficient execution in power-stringent platforms [2][22] The reduction of conditional statements improves branch predictions and achieves better cache performance. The reduction of attributes of classes simplifies the runtime stack of programs and decreases the memory footprint of objects.

Implementation	*Size*	*CC*	*WC*	*EC*
Original	23277	3.69	2404	2423
Re-factored	13524	3.05	1549	1899
Reduction	9753	0.64	855	525

Table 4: Reduction of Overall Structure. (CC: Cyclomatic complexity. WC: Weight of Class. EC: Efferent coupling).

6.2 Performance Evaluation

This section presents the benchmarking results collected on three versions of the OBRacus implementation: the refactored ORBacus core with aspects taken out (None), the original implementation (Original), and the combined implementation with all aspects "woven" back in via the AspectJ compiler (All). We use the Open CORBA Benchmarking Suite (OCBS) [33] to provide a thorough comparison of the runtime performance of these three versions. OCBS measures the performance of the following CORBA functional areas: invocation, marshalling, dispatching, parallelism, as well as combinations of these areas. All of the benchmarking operations are supported without modification by both the original ORBacus implementation and our refactored version at almost half of the original size. The tests are performed on Pentium 4 2GHZ running Redhat 8.0 with 1G of memory.

[22]Although the paper's analysis is based on Java systems, we believe, that our discussion is not limited to Java systems either.

Aspect	*Arguments*	*Conditional*	*Attributes*
`Any` & TypeCode	0	8	2
Encoding Conversion	6	9	9
Oneway Call	8	7	1
Wchar & Wstring	4	44	8
Total	18	68	20

Table 5: Reduction of Code Elements Caused by Crosscutting.

Implementation	*Median*	*Average*
None	157	203
Original	167	221
All	180	238

Table 6: Invocation Cost in microseconds.

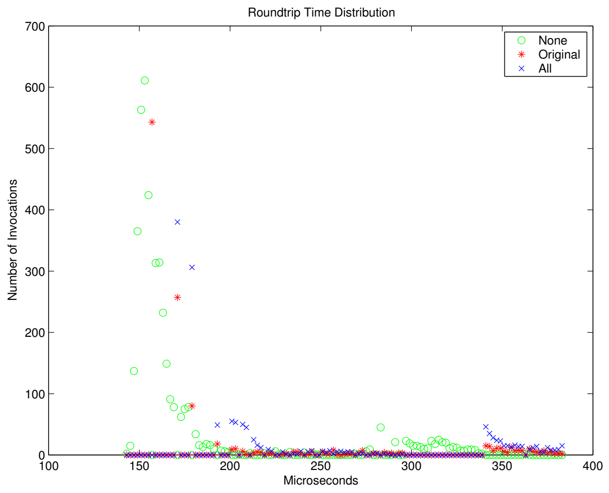


Figure 9: Invocation Time Distribution in microseconds.

6.2.1 *Invocation Roundtrip*

The invocation round-trip is to measure how fast can message traverse the middleware stack excluding the network delay. Table 6 presents the median and average invocation time (microsecond) observed by OCBS with a client invoking a "do-nothing" service on the server. The numbers are computed as averages of over 4,500 sample invocations. It shows that the refactored core achieves approximately 8% performance gain on average. The combined implementation (All) incurs 7.6% invocation overhead. Figure 9 compares the histograms of the round trip delays for all three versions. It can be observed that a large majority of the round-trips made over the refactored version densely concentrate within the 150-175 microsecond interval. Distributions for the original version and the "woven" version (All) are primarily beyond the 160 mark and largely the same. This shows the invocation time of the refactored core is also statistically better than the original.

6.2.2 *Data Transport*

In this category, OCBS measures the time taken by CORBA to transport a certain amount of data between the client and the server. The weighting core functionality in this test is the efficiency of marshalling and unmarshalling. Figure 10 shows the performance measurement of a client sending a stream of octets to the server. Figure 11 shows the reverse communication with the same input parameters. The X-axis denotes the number of octets and the Y-axis the average invocation time. The samples for these graphs, as well as for all graphs hereafter, are the round-trip times for all invocations made in a period of 10 seconds. Our observation is that, for the best possible performance Figure 10(1), the refactored core is equivalent to the original core. On average, the refactored core performs slightly better than the original. In the minimum and the average case, all three versions are largely equivalent. The refactored core performs the best and the combined performs the worst in the majority of cases. This improvement is a combined effect of: 1. a lighter-weight marshalling/unmarshalling layer supporting a fewer number of CORBA data types; 2. a simpler marshalling/unmarshalling logic with no need to decide on character encoding schemes and to setup a proper conversion mechanism. The performance of the "All" version shows that configuring these functionalities back into the marshalling/unmarshalling mechanisms does not incur significant overhead.

6.2.3 Request Dispatch

Figure 12 present the evaluation for the invocation cost in the presence of multiple server objects. This reflects the effect of server side call dispatching mechanisms. The X-axis is labeled by the number of instantiated objects. The Y-axis represents the invocation time. As the number of server objects increases, the average invocation time also increases in all versions. We observe a similar pattern, as the refactored core (None) is most efficient in request dispatching, as compared to the original implementation. This is not surprising since the dispatching logic is simplified by taking out decisions on dispatching both dynamic requests and local invocations. The combined version, in the best (Min) and average scenarios, exhibits a penalty between 5% to 13% as a result of AspectJ's bytecode "weaving". In the worse-case scenario (Max), the dispatching performance of all three versions are equivalent.

6.2.4 Parallel Execution

Figure 13 measures the performance of remote invocations in the multi-threaded scenario. The X-axis is the number of threads created on the client side and the Y-axis the invocation time in microseconds. The multi-threaded performance agrees with the single threaded invocation performance with the refactored core performs the best, the original in the medium range, and the combined the worst. The overall mechanism of downcall in the refactored core is much simpler and lighter compared to the original implementation, which contributes to its improved performance. Moreover, refactoring orthogonal functionalities away from the core reduces the shared data among threads. This shared data originally exists in process-wide singletons and includes codeset factories, conversion utilities, default dynamic servers, just to name a few. Therefore, the overhead for inter-thread communication is reduced.

6.2.5 Combined Execution

In Table 7, we show the performance evaluated by combining multiple servers and multiple threads and exchanging messages of size 50K octets between the client and the server. The benchmarking tool measures the following scenarios: A. Message sending using 100 client threads; B. Message receiving using 100 client threads; C. Remote invocations (Ping) by 100 client threads to 50,000 servers; D. Message sending to 50,000 servers; E. Message receiving from 50,000 servers; F. Message sending by 100 client threads to 50,000 servers; G. Message receiving of 100 client threads from 50,000 servers.

Except for scenarios A and F, the refactored ORB generally performs the best although the differences among all three versions are typically within 5%. Scenarios A and F are anomalies possibly caused by the OS scheduler. Similar anomalies are also noted by the original authors of the benchmark [33].

	A	*B*	*C*	*D*	*E*	*F*	*G*
Re-factored	7036	2161	193	954	1061	6020	2689
Original	2332	2395	199	976	1055	1686	3130
Combined	6142	2366	238	1003	1103	5396	3111

Table 7: Combined Benchmarking Results in microseconds.

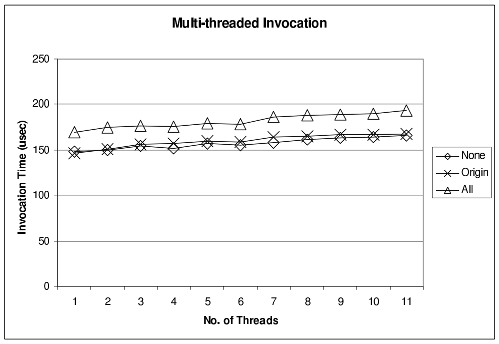


Figure 13: Cost of Using Multiple Threads in microseconds.

6.2.6 Cache Performance

Another important metric for evaluating the runtime effect of the horizontal decomposition method is the cache behavior of the system. We use the performance counter library[23] to count various microprocessor events. These measurements are conducted by using a simple loop to send a long integer to a remote server. The entire data collection period consists of 100 epochs, where each epoch equals 1500 remote method calls conducted with ORBacus. Table 8 depicts the cache-miss rates computed as the average over 100 epochs. A decrease of the instruction-cache miss rate is an indicator of a simpler control flow. Better L2[24] cache performance represents better locality and a higher degree of cohesion in the program. The data shows that, to support the same functionality, the combined version adds a slight overhead. The refactored core performs better than the original version. This is consistent with our previously presented benchmarking results.

	None	*Original*	*All*
L1 Instruction Misses	368869	380404	404187
L2 Miss Rate	3.25%	3.83%	4.4%

Table 8: Middleware Cache Measurements: L1 and L2.

[23]`http://www.fz-juelich.de/zam/PCL/`

[24]L2 miss rate is calculate by number of cache misses divided by the total number of load/store instructions in that epoch

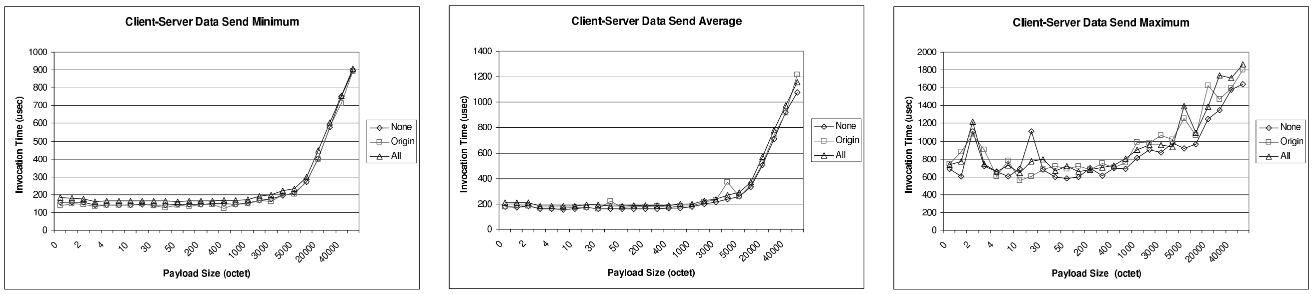


Figure 10: Data Sending Client to Server in microseconds. (1) Minimum Cost (2) Average Cost (3) Maximum Cost.

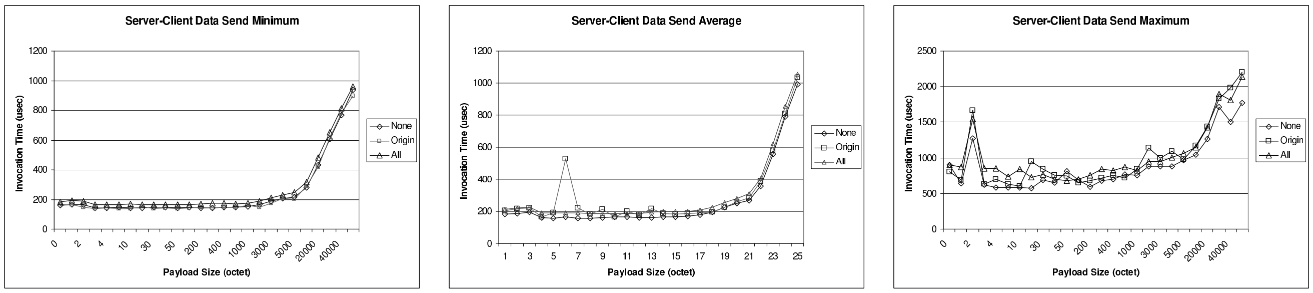


Figure 11: Data Sending Server to Client in microseconds. (1) Minimum Cost (2) Average Cost (3) Maximum Cost.

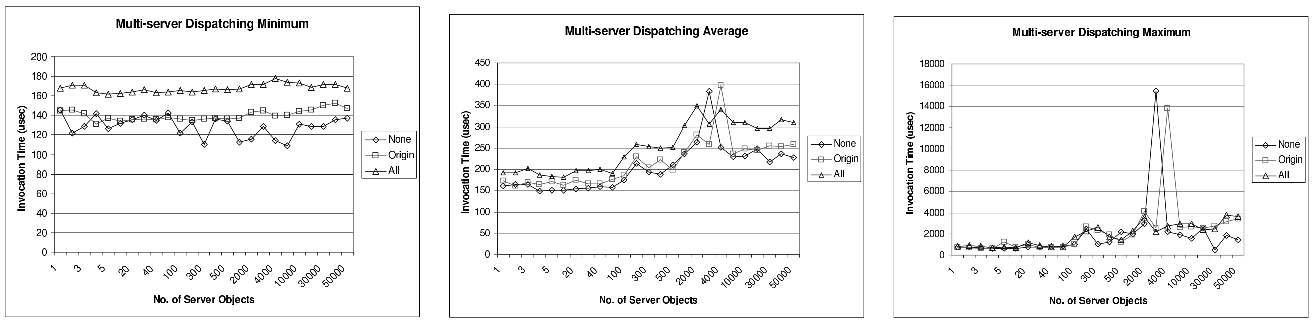


Figure 12: Dispatching to Multiple Servers in microseconds (1) Minimum Cost (2) Average Cost (3) Maximum Cost.

6.2.7 Concluding Remarks

In conclusion, the benchmark data clearly indicates that, for the primary ORB functionalities, ORBacus achieves good performance gains with orthogonal features separated into the aspect code. Although this improvement is not as dramatic as we previously anticipated, it validates that horizontal decomposition principles are effective in separating convoluted features from the middleware core without compromising its functionality. We expect to observe more dramatic improvements with continued refactoring efforts. Horizontal decomposition is applicable to any middleware implementation, and more generally, any software system, yet we are limited by the maturity of aspect oriented programming languages available to date. Choosing AspectJ and "aspectizing" applications running on virtual machines might be another contributing factor for less dramatic speed-ups[25]. We are not too concerned with the small overhead in some of the results induced by combining aspects. This is because negligible overhead is one of the most vital design objectives of the AspectJ compiler. We expect research in more efficient code generation and JIT techniques for AspectJ will eventually amortize this cost. We defer the benchmark comparison for the aspectual functionality to future work since the required benchmarking options are not available to us in the benchmarking tool. Our past work, however, has shown that features supported through aspects do not experience significant runtime overhead using AspectJ [41].

7. RELATED WORK

There is a broad range of research related to horizontal decomposition principles, or, more generally, improving the configurability and the adaptability of software systems through the use of new modularization techniques. We present the related research in three categories: existing aspect oriented applications to middleware, adaptive and customizable middleware, and feature oriented programming.

7.1 Existing Middleware Applications of AOP

A large number of current applications of aspect oriented programming to middleware architectures focus on providing better modularization and support for QoS properties, or, more broadly speaking, support for non-functional properties in general. The QuO project at BBN Technologies constitutes a framework supporting the development of distributed applications with QoS requirements (see [26, 12], for example). QuO uses quality description languages (QDL) to specify client-side QoS needs, regions of possible level of QoS, system conditions that need to be monitored, certain behavior desired by clients, and QoS conditions. Loyall *et al.* [26] interpret these different description languages as aspect languages that are processed by a code generator to assembled a runtime environment supporting the desired and expected quality of service by client and server in a distributed application. The COMQUAD [16] project uses a similar approach to addresses non-functional aspects in component-based systems. CQML$^+$ is an XML-based language in COMQUAD for the description of "quality characteristics". This description is then interpreted and maintained by the component containers. The JBoss application server[26] provides interceptors to allow the hosted applications to handle crosscutting concerns. It also directly supports common crosscutting concerns in J2EE application servers such as persistence, security, and transaction. Colyer and Clement [8] demonstrate how to apply aspect orientation techniques in an industrial setting to refactor a major crosscutting concern from an application server. Hunleth *et al.* [19], as well as its extended work, FACET[27], take a similar position as we do and aim at customizing middleware with aspect oriented techniques. It is suggested that aspects could be used for consistency checking, error handling, and at the interface specification level. None of the concepts suggested appears to be evaluated, so a comparison with our approach is difficult. Similarly, Jacobsen and Krämer [20] have suggested to expose certain crosscutting concerns at the interface level to make them pluggable on a by-need basis.

Our research differs from many of the above approaches by focusing on the customizability of the middleware mechanism itself rather than modularizing its extrinsic properties. The crosscutting problems we attack are not limited to non-functional concerns but more generally apply to any orthogonal functionality of the architecture. Moreover, unique to our approach is that we introduce and evaluate a set of principles to guide the aspect oriented design of systems and the refactoring of such systems.

7.2 Adaptive Middleware

The discussion of separation of concerns for the design of middleware platforms can be broadly classified into approaches that provide *customization* of the middleware through static or dynamic policy selection [1, 34], approaches that adapt the operation of the middleware to changing runtime conditions through the use of reflection [7, 25, 24, 5], and approaches based on various forms of aspect definitions and interpretations [26, 32, 19]. Many of these projects use several of these techniques in combination. Below, we discuss each category in turn and point out how our approach is distinguished. Upfront we can say that the key differentiator in our approach is the focus on a methodology to design flexible and customizable software systems in general and middleware in particular and the evaluation of this methodology through extensive aspect oriented refactoring of a legacy middleware platform. Our overall emphasize is re-designing and re-structuring the system implementation based on the emerging aspect oriented development paradigm. Astley *et al.* [1] achieve middleware customization through techniques based on separation of communication styles from protocols and a framework for protocol composition. The CompOSE—Q [34] project uses an actor-based model for runtime adaptation. Both approaches do not employ aspect orientation to isolated crosscutting design concerns from the middleware implementation.

Several projects exploit reflective programming techniques to allow the middleware platform to adapt itself dynamically to changing runtime conditions [7, 25, 24, 5]. This includes projects such as openORB [7], openCORBA [25], dynamicTAO [24], the OpenOrb project [5], and also the CompOSE—Q project [34]. In these approaches, the reflective middleware implementation observes and reacts to changing

[25]A measure of raw socket performance on the same machine shows a 90 microsecond roundtrip for sending 1K of data.

[26]JBoss URL:`http://www.jboss.org`

[27]FACET URL: `http://www.cs.wustl.edu/~doc/RandD/PCES/facet/`

environmental conditions by selecting different implementation strategies. The platform adapts itself to the environment, but is not customized to domain or application requirements. LegORB[28], the Universally Interoperable Core (UIC)[29], and Jonathan[30] are customizable middleware platforms. Customization ranges from selecting the transport protocol to method dispatching and marshalling routines. While these approaches focus on customizing key platform functions, they do not concern themselves with the actual implementation and whether or not the function is indeed a crosscutting concern. Moreover, the customization focuses on a coarse level. In our approach, customization is much finer-grained, allowing individual types to be separated from the middleware implementation.

7.3 Feature Oriented Programming

Feature oriented programming [29] is an alternative programming paradigm for increasing the flexibility of conventional inheritance-based typing in object oriented systems. In FOP, base objects, features which "crosscut" base objects, and the interactions between features and base objects exit in separate modules. Hindsight shows some properties of FOP can also be identified in our AOP implementation. The concept of feature interaction bears similarities to the implementation convolution problem. However, the former is an intended engineering principle of FOP whereas the latter describes an unintended phenomenon in legacy implementations. Moreover, like AOP, guidelines are still needed as how FOP can be applied to improve the middleware architecture. The idea of separating the core from features can also be found in research on telecommunication systems [40] where the relationship is discussed at the level of system functionality not at the architectural level.

8. CONCLUSION

Distributed applications are becoming more and more reliant on the middleware layer, which decouples them from the complexity of distributed application development. The increasing heterogeneity and versatility of application domains requires middleware to support an unprecedented level of configurability and adaptability. We believe this level is difficult to achieve with vertical decomposition methods alone, due to their inability in maintaining convolution-free implementations. Hence, we propose the horizontal decomposition principles and advocate the mixed-paradigm architecture of middleware. This is based on using the conventional hierarchically-decomposed architecture to support core operations and aspect oriented decomposition for adding orthogonal properties. In the horizontal decomposition, we emphasize the relative nature of aspects and the importance of defining the core decomposition as the basis of the AOP decomposition. Through horizontal decomposition, we have made two major improvements as compared to the conventional architecture:

1. We have made considerable progress in factoring out major middleware functionality as aspects and have obtained a stripped-down version of the middleware core. This core is 40% of its original size, and its performance has improved on all of the third-party benchmarks. This is not at the cost of compromising overall functionality: the "stripped-out" properties can be transparently brought into, or taken out of the middleware on a by-need basis through the "weaving" mechanism of AspectJ.
2. We have dramatically increased the degree of configurability and adaptability of the middleware by resolving the convolution among aspects and making the architecture super-impositional. We have turned a monolithic architecture into an architecture with over 60 possible combinations of features[31]. These combinations are composed at the post-compilation stage at which point no source code modification is required.

Although our primary experiments with horizontal decomposition focus on middleware, we believe that our approach, of using multiple decomposition paradigms to untangle and to separate the architectures of both core operations and orthogonal functionalities, is generally applicable to any software architecture.

We suspect that horizontal decomposition principles are best suitable for applications that serve multiple domains and support a wide range of usage scenarios. The successful application of horizontal decomposition is based on discerning the most common functionality, i.e., the core, from a large and complex system. We are somewhat blessed in that the requirements and the functionalities of most middleware technologies, such as CORBA, are well-studied. Software architecture in other areas might not have this advantage. Another issue, which is out of the scope of this paper but cannot be neglected, is the question of how to ensure consistency across aspects. That is to ensure that functionality in one aspect does not negate or indirectly change the semantics of another aspect. We had not encountered this problem due to the refactoring of an existing application. However, this becomes a serious issue when developing new applications or adding new aspects. Besides intuition and heuristics, this problem could be addressed, either at the requirements engineering level, or through the use of model checking. It is also a challenge to manage various configurations, since the possibilities of combinations of aspects to form new product versions grows exponentially with the number of aspects. An effective configuration tool must accompany the architecture to assist the customization process.

In our future work, we will continue to accumulate more experience in applying the horizontal design principles to improve the modularity of middleware. We will continue our refactoring work to separate out more horizontal design concerns from the middleware core. For example, we can further decompose primitive IDL data types as aspects. The interface compiler, such as the IDL compiler in CORBA, is often an integral part of the middleware functionality. We have started studying the interface compiler support for horizontal decomposition. We will also explore the application of the horizontal decomposition method more extensively by experimenting with other middleware types such as J2EE application servers. This will greatly assist us in designing

[28]`http://choices.cs.uiuc.edu/2k/LegORB/`

[29]http://www.ubi-core.com/

[30]http://www.objectweb.org/jonathan/jonathanHomePage.htm

[31]A rough calculation: the number of combinations of 6 aspects: interceptor support, DII, DSI, local invocation, wide characters, conversion support is 2^6=64.

a fully aspect oriented middleware platform, which is our long term objective.

Acknowledgments

This research has been supported in part by an NSERC grant and in part by an IBM CAS fellowship for the first author. The authors are very grateful for this support.

9. REFERENCES

[1] M. Astley, D.C. Sturman, and G. A. Agha. Customizable Middleware for Modular Software. *ACM Communications*, May 2001.

[2] Luca Benini and Giovanni De Micheli. System-Level Power Optimization: Techniques and Tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, 2000. p173.

[3] G. D. Bergland. Structured Design Methodologies. In *Proceedings of the No 15 Design Automation Conference on Design Automation*, pages 475–493. IEEE Press, 1978.

[4] L. Bergmans and M. Aksit. Aspects and crosscutting in layered middleware systems. Reflective Middleware (RM 2000) workshop held in conjunction with the IFIP/ACM Intl. Conf. on Distributed System Platforms and Open Distributed Processing (Middleware 2000)., April 2000.

[5] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online Journal 2(6)*, 2001.

[6] Joey Caron, Scott Herscher, and Ann Marie O'Connor. CORBA in the palm of your hand whitepaper. Vertel Corporation.

[7] M. Clarke, G. Blair, G. Coulson G., and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'2001)*, November 2001.

[8] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *3rd International Conference on Aspect-oriented Software Development (AOSD'04)*, pages 56 – 65, Lancaster, UK, 2004.

[9] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[10] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[11] Louis DiPalma and Robert Kelly. Applying CORBA in a contemporary embedded military combat system. OMG's Second Workshop on Real-time and Embedded Distributed Object Computing, June 2001.

[12] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building adaptive distributed applications with middleware and aspects. In *Proceedings of the 3rd international conference on aspect oriented software development*. ACM, 2004.

[13] Robert Filman. Achieving ilities. URL: `http://ic.arc.nasa.gov/~filman/text/oif/wcsa-achieving-ilities.pdf`, 1999.

[14] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[16] Steffen Gobel, Christoph Pohl, Simone Rottger, and Steffen Zschaler. The COMQUAD Component Model: Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects. In *Proceedings of the 3rd International Conference on Aspect Oriented Software Development*. ACM, 2004.

[17] Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical report, December 2001.

[18] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173. ACM Press, 2002.

[19] Frank Hunleth, Ron Cytron, and Christopher Gill. Building customizable middleware using aspect oriented programming. In *Workshop at OOPSLA*, 2001.

[20] H.-A. Jacobsen and B. J. Krämer. A design pattern based approach to generating synchronization adaptors from annotated IDL. In *IEEE Automated Software Engineering Conference (ASE'98)*, pages 63–72. IEEE Computer Society, September 1998.

[21] Hans-Arno Jacobsen. Middleware architecture design based on aspects, the open implementation metaphor and modularity. In *Workshop on Aspect-Oriented Programming and Separation of Concerns*, Lancaster, UK, August 2001.

[22] Mik Kersten and Gail C. Murphy. Atlas: a case study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the 14th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 340–352. ACM Press, 1999.

[23] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[24] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[25] Thomas Ledoux. OpenCorba: a reflective open broker. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 197–214, Saint-Malo, France, July 1999. Springer-Verlag.

[26] Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson. QoS aspect languages and their runtime integration. In *Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers. Lecture Notes in Computer Science, Vol. 1511, Springer-Verlag*, Pittsburgh, Pennsylvania, USA, May 28th-30th 1998.

[27] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155. ACM Press, 1987.

[28] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–58, December 1972.

[29] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, volume 1241 of

Lecture Notes in Computer Science, page 419 ff, 1997.

[30] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley & Sons, Ltd, 1 edition, 1999.

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

[32] L. Teboul, R. Pawlak, L. Seinturier, E. Gressier-Soudan, and E. Becquet. AspectTAZ: A new approach based on aspect-oriented programming for object-oriented industrial messaging services design. In *WFCS 2002*, August 2002.

[33] Petr Tuma and Adam Buble. Open CORBA Bench Marking. *SPECTS 2001*. URL: `http://nenya.ms.mff.cuni.cz/~bench`.

[34] Nalini Venkatasubramanian, Mayur Deshpande, Shivjit Mohapatra, Sebastian Gutierrez-Nolasco, and Jehan Wickramasuriya. Design & implementation of a composable reflective middleware framework. In *IEEE International Conference on Distributed Computer Systems (ICDCS-21)*, April 2001.

[35] Robert J. Walker, Elisa L.A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *In Proceedings of the 21st International Conference on Software Engineering*, pages 120–130, 1999.

[36] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, 30(4):459–527, 1998.

[37] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. Dado: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *Proceedings of the 25th International Conference on Software Engineering*, pages 174–186. IEEE Computer Society, 2003.

[38] Edward Yourdon and Larry L. Constantine. *Structured Design*. Prentice-Hall, Inc., 1979.

[39] Pamela Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 6(5):15–25, September 1989.

[40] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–29, August 1993.

[41] Charles Zhang and Hans-Arno Jacobsen. Quantifying Aspects in Middleware Platforms. In *2nd International Conference on Aspect Oriented Systems and Design*, pages 130–139, Boston, MA, March 2003.

[42] Charles Zhang and Hans-Arno Jacobsen. Refactoring Middleware Systems: A Case Study. In *International Symposium on Distributed Objects and Applications (DOA 2003)*, Catania, Sicily (Italy), 2003. Lecture Notes in Computer Science, Springer Verlag.

[43] Charles Zhang and Hans-Arno Jacobsen. Refactoring Middleware with Aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, November 2003.

[44] Charles Zhang and Hans-Arno Jacobsen. TinyC^2 — towards a dynamic weaving aspect language based on C. In *Foundation of Aspect Oriented Languages (FOAL) jointly held with the 2nd International Conference on Aspect Oriented Systems and Design*, Boston, MA, March 17th 2003.