Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures*

Arvind S. Krishna†, Aniruddha Gokhale† and Douglas C. Schmidt†,
Venkatesh Prasad Ranganath‡ and John Hatcliff‡
†Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN
‡Dept. of Computing and Information Sciences, Kansas State University, Manhattan, KS

ABSTRACT

Product-line architectures (PLA)s are an emerging paradigm for developing software families for distributed real-time and embedded (DRE) systems by customizing reusable artifacts, rather than handcrafting software from scratch. To reduce the effort of developing software PLAs and product variants for DRE systems, developers are attempting to leverage general-purpose – ideally standard – middleware platforms whose reusable services and mechanisms support a range of application quality of service (QoS) requirements, such as low latency and jitter. The generality and flexibility of standard middleware, however, often results in excessive time/space overhead for DRE systems, due to lack of optimizations tailored to meet the specific QoS requirements of different product variants in a PLA.

This paper provides the following contributions to the study of middleware specialization techniques for PLA-based DRE systems. First, we identify key dimensions of generality in standard middleware, including generality stemming from framework implementations, deployment platforms, and middleware standards. Second, we illustrate how context-specific specialization techniques can be automated and applied to tailor standard middleware to better meet the QoS needs of different PLA product variants. Third, we quantify the benefits of applying automated tools to specialize a standard Real-time CORBA middleware implementation. When applied together, these middleware specializations improved our application product variant throughput by ∼65%, average- and worst-case end-to-end latency measures by ∼43% and ∼45%, respectively, and predictability by a factor of two over an already optimized middleware implementation, without affecting portability, standard middleware APIs, or application software implementations, and preserving interoperability wherever possible.

1. INTRODUCTION

Emerging trends & challenges. *Product-line architectures* (PLA-s) [17, 1] are a promising technology for systematically addressing the challenges of large-scale software systems. In contrast to conventional software processes that produce separate point solutions, PLA-based processes create families of *product variants* [30] that share a common set of capabilities, patterns, and architectural styles. PLAs can be characterized using *scope, commonality, and variabilities* (SCV) analysis [2], which is an engineering process that identifies the scope of the product families in an application domain and then determines the common and variable properties among them.

PLAs have been created and applied to a variety of domains [24, 9], including the domain of *distributed, real-time and embedded (DRE) systems* [30, 5, 31]. Examples of DRE systems include applications with hard real-time requirements, such as avionics mission computing [26], as well as those with softer real-time requirements, such as telecommunication call processing and streaming video [20]. The QoS challenges (such as low memory footprint, latency, and predictability) of DRE systems have hitherto led developers to (re)invent custom applications that are tightly coupled to specific hardware/software platforms, which is tedious, error-prone, and costly to evolve over product lifecycles. During the past decade, therefore, a key technology for alleviating the tight coupling between applications and their underlying platforms has been *middleware*, which (1) functionally bridges the gap between applications and platforms, (2) controls many aspects of end-to-end QoS, and (3) simplifies the integration of components developed by multiple technology suppliers.

Although middleware has been used successfully in DRE systems [30, 5, 31, 26], key challenges must be overcome before it can be applied broadly to support the QoS needs of *PLA-based* DRE systems. In particular, R&D is needed to help resolve the tension between (1) the *generality of standards-based middleware platforms*, which benefit from reusable architectures designed to satisfy a broad range of application requirements, and (2) *application-specific product variants*, which benefit from highly-optimized, custom middleware implementations. In resolving this tension, solutions should ideally retain the portability and interoperability afforded by standard middleware.

Specializing Middleware for PLAs. This paper extends our earlier work on general-purpose middleware optimizations [20, 21, 23] by describing the results of developing and applying a toolkit to help resolve key aspects of the generality/specificity tension outlined above. This toolkit automates the *specialization* [4] of general-purpose, standards-based middleware to meet the needs of specific PLA-based DRE systems. This paper provides the following research contributions:

1. We use a representative PLA case study to identify key dimensions of *excessive generality* in standards-based middleware, focusing on Real-time CORBA [15], which is standard middleware used in Boeing Bold Stroke [30, 31, 26], which is a PLA-based DRE system in the domain of avionics mission computing.
2. We show how *context-specific specialization techniques* [10]

*This research was funded by grants from DARPA and Qualcomm.

(such as code refactoring [6], and code weaving [32]) can be used to customize the widely used TAO [28] Real-time CORBA implementation to remove excessive generality and thus better support application-specific QoS needs of PLA-based DRE systems, such as Bold Stroke.

3. We describe the design of a domain-specific language, tools, and a process for *automating the specialization techniques* discussed in the paper.
4. We present and analyze *quantitative results* that demonstrate the improvement in performance and predictability of specializations applied to TAO in the context of our PLA case study.

Our results show that specialization techniques guided by context-specific information can significantly improve the QoS of a standards-based middleware implementation that has already been optimized extensively via general-purpose techniques.

Paper organization. The remainder of the paper is organized as follows: Section 2 identifies key middleware specialization challenges pertaining to PLA-based DRE systems; Section 3 explains how context-specific specializations of middleware help address these challenges and describes our techniques and tools for automating the specializations; Section 4 examines the results of experiments conducted to validate our approach; Section 5 compares our work with related research; and Section 6 presents concluding remarks and summarizes lessons learned.

2. MIDDLEWARE SPECIALIZATION CHALLENGES

General-purpose, standard middleware implementations are designed to be reusable since they need to satisfy a broad range of functional and QoS application requirements. PLAs define a family of systems that have many common functional and QoS requirements, as well as variability specific to particular products built using the PLA. Resolving the tension between generality and specificity is essential to ensure that middleware can support the QoS requirements of PLA-based DRE systems. Unfortunately, implementations of standards-based, QoS-enabled middleware, such as Real-time CORBA and Real-time Java, can incur time/space overheads due to excessive generality.

This section uses a representative PLA-based DRE system scenario to identify and illustrate common types of excessive generality in standard middleware. Section 3 then outlines how context-specific middleware specialization techniques and tools help to alleviate the time/space overhead stemming from this generality.

2.1 DRE PLA Case Study

This section uses a concise, yet representative, DRE PLA scenario to (1) illustrate how the generality/specificity tension outlined above occurs in production DRE systems and (2) identify concrete system invariants that drive our specialization approach. The scenario is based on the Boeing Bold Stroke avionics mission computing PLA [31], which supports the Boeing family of aircraft, including many product variants, such as F/A-18E, F/A-18F, F-15E, F-15K, etc. Bold Stroke is a component-based, publish/subscribe platform built atop the TAO Real-time CORBA ORB.

Figure 1 illustrates the *BasicSP* application scenario, which is an assembly of avionics mission computing components reused in different Bold Stroke product variants and representative of rate-based DRE systems in avionics, vetronics, and process control. This scenario involves four avionics mission computing components that periodically send GPS position updates to a pilot and navigator cockpit displays at a rate of 20 Hz. The time to process inputs to the system and present output to cockpit displays should therefore be less than a single 20 Hz frame.

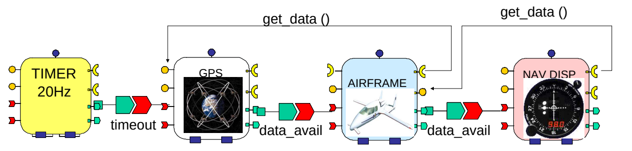

Figure 1: *BasicSP* Application Scenario

Communication between components uses the event-push/data-pull model, with data producing components pushing an event to notify new data is available and data consuming components pulling data from the source. A `Timer` component pulses a `GPS` navigation sensor component at a certain rate, which in turn publishes the `data_avail` events to an `Airframe` component. Aware that new data is available, this component then calls a method provided by the `Read_Data` interface of the `GPS` component to retrieve the current location. After formatting the data, `Airframe` sends a `data_avail` event to the `Nav_Display` component, which then pulls the location and velocity data from the `Airframe` component and displays this information on the pilot's heads-up display.

The *BasicSP* scenario illustrates a range of commonalities and variabilities in the Bold Stroke PLA. *Commonalities* include the set of reusable components (such as `Display`, `Airframe`, and `GPS`) in Bold Stroke and middleware capabilities (such as connection management, data transfer, concurrency, synchronization, (de)marshaling, (de)multiplexing, and error-handling) that occur in all product variants. *Variabilities* include application-specific component connections (such as how `GPS` and `Airframe` components are connected in an F/A-18E vs. an F-15K), different implementations (such as whether GPS or inertial navigation algorithms are used), and components specific to particular customers (such as restrictions on exporting certain encryption algorithms). The rates at which these components interact is yet another variability that may change in different product variants.

Analysis of commonalities and variabilities in the *BasicSP* scenario helps identify *functional* (*e.g.*, specific communication protocols) and *QoS* (*e.g.*, end-to-end latency) characteristics of PLAs. In turn, these characteristics map to specific requirements on – and potential optimizations of – the underlying middleware. The remainder of this paper focuses on the specialized middleware optimizations based on PLA functional and QoS characteristics.

2.2 Common Types of Excessive Generality in Middleware

We now identify and describe key types of excessive middleware generality that are relevant to PLA-based DRE systems. We use the *BasicSP* scenario from Figure 1 to show how this generality manifests itself in a PLA-based DRE system. The challenges of each type of generality are shown in Figure 2 and discussed below.[1]

Challenge 1. Overly extensible object-oriented frameworks. Middleware is often developed as a set of object-oriented frameworks that can be extended and configured with alternative implementations of key components, such as different types of transport protocols (*e.g.*, TCP/IP, VME, or shared memory), event demultiplexing mechanisms (*e.g.*, reactive-, proactive-, or thread-based), request demultiplexing strategies (*e.g.*, dynamic hashing, perfect

[1]These challenges are not limited to Real-time CORBA or the *BasicSP* PLA, which we use simply as examples to make the discussion concrete.

hashing, or active demuxing), and concurrency models (*e.g.*, thread-per-connection, thread pool, or thread-per-request). A particular DRE product variant, however, may only use a small subset of the potential framework alternatives. As a result, general-purpose middleware may be *overly* extensible, *i.e.*, contain unnecessary overhead for indirection and dynamic dispatching that is not needed for use cases in a particular context.

In the *BasicSP* scenario, for instance, the transport protocol is VME, the event demultiplexing mechanism is reactive, the request demultiplexing mechanisms are perfect hashing and activate demuxing, and the concurrency model is thread pool. A different variant of this scenario for a different set of customer requirements, however, may use a different set of framework components. *A challenge is to develop middleware specialization techniques that can eliminate unnecessary overhead associated with overly extensible object-oriented framework implementations for certain product variants or application-specific contexts.*

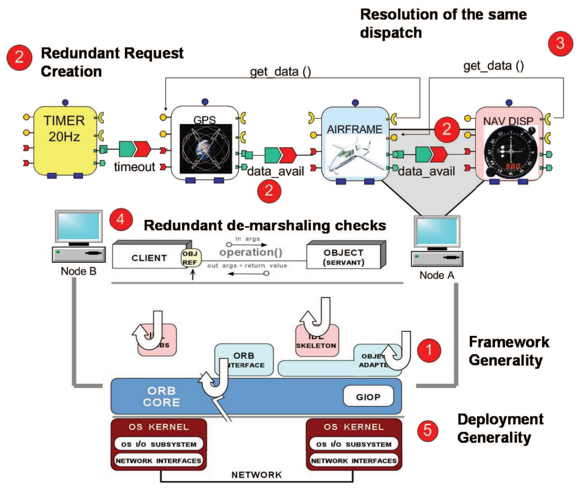

Figure 2: *BasicSP* Specialization Points

Challenge 2. Redundant request creation and/or initialization. To send a request to the server, the middleware creates a *request*, which contains buffer space to hold the header and payload information for each invocation. Rate-based DRE systems often repeatedly generate periodic events, such as timeouts that drive periodic system execution. Since most request information (such as message size, operation name, and service context) does not change across events, middleware implementations can use buffer caching [20] strategies to minimize request creation. This approach, however, can still incur the overhead of initializing the header and payload for every request.

In the *BasicSP* scenario, for instance, the `Timer` component always sends the same `timeout` event to the `GPS` component. Similarly, the `GPS` and `Airframe` components send the same `data_avail` event to their consumers. A different variant of this scenario, however, may not send the same events to consumers repeatedly. *A challenge is to develop middleware specialization techniques that can reuse pre-created requests (i.e., from previous invocations) partially and/or completely to avoid redundant initialization for certain product variants or application-specific contexts.*

Challenge 3. Repeated resolution of the same request dispatch. To minimize the time/space overhead incurred by opening multiple connections to the same server, middleware often multiplexes requests on a single connection between client and server processes. Multiple client requests targeted for different request handlers in a server process are therefore received on the same multiplexed connection. Standard Real-time CORBA servers typically process a client request by navigating a series of middleware layers, *e.g.*, ORB core, object adapter(s), servant, and operation. To optimize request demultiplexing, Real-time CORBA implementations can combine active demultiplexing [20] and perfect-hashing [20] strategies to bound worst case lookup time to $O(1)$, irrespective of the nesting of the layers. This optimization, however, can still incur non-trivial overhead when navigating middleware layers and is redundant when the target request handler remains the same across different request invocations.

In the *BasicSP* scenario, for instance, the `Airframe` and `Nav_Display` components repeatedly use the same `get_data()` operation to fetch new GPS and Display updates. In a connection between `GPS` and `Airframe` components, therefore, the `get_data()` operation is sent and serviced by the same request dispatcher. A different variant of this scenario, however, may service operations via different request dispatchers. *A challenge is to develop middleware specialization techniques that avoid the expense of navigating layers of middleware to resolve the same request dispatch for certain product variants or application-specific contexts.*

Challenge 4. Redundant (de)marshaling overheads. PLA-based DRE systems may be deployed on platforms with different instruction set byte orders. To support interoperable request processing regardless of byte order, standard Real-time CORBA implementations therefore use the General Inter-ORB Protocol (GIOP), which performs byte order tests when (de)marshaling requests/responses. These tests incur unnecessary overhead, however, when all the DRE system computing nodes have the same byte order. The GIOP protocol also requires alignment of primitive types (such as `long` and `double`) within a request/response for certain hardware architectures, which forces middleware implementations to maintain offset information within a request/response buffer and pad buffers up to the next readable/writable locations. Frequent alignment and padding can force costly buffer resizing and data copying. The overhead associated with alignment can be eliminated in homogeneous environments, *i.e.*, when the same ORB implementation and compiler are used for (de)marshaling.

In the *BasicSP* scenario, for instance, the two nodes on which the components are deployed (*NodeA* and *NodeB*) have the same byte order. The standard TAO Real-time CORBA middleware residing on these nodes, however, still tests whether (de)marshaling is needed when requests/responses are exchanged between nodes. A different variant of this scenario, however, may run on nodes with different byte orders, but with the same compiler/middleware implementation, in which case data need not be aligned. *A challenge is to develop middleware specialization techniques that evaluate ahead-of-time deployment properties to remove redundant (de)marshaling overheads for certain product variants or application-specific contexts.*

Challenge 5: Generality of deployment platform. Another key dimension of generality stems from the deployment platforms on which middleware and PLA applications are hosted. Examples of this deployment platform generality include different OS-specific system calls, compiler flags and optimizations, and hardware instruction sets. Every OS, compiler, and hardware platform provides different configuration settings that perform differently and can be tuned to minimize the time/space overhead of middleware and applications.

In the *BasicSP* scenario, for instance, a product variant could run the Linux OS with Timesys kernel and g++ compiler on *NodeA* and the VxWorks OS with the Greenhills compiler on *Node B*. Yet other variants could use different combinations of OS, complier, and hardware. *A challenge is to develop specialization techniques that discover and automate the selection of right combination of OS, compiler, and hardware settings for a given deployment platform.*

2.3 Summary

This section described key dimensions of excessive middleware generality, using Real-time CORBA middleware as an example. These challenges are also applicable to other popular middleware platforms that use common patterns [7, 29] to accommodate PLA variability, such as different protocols, concurrency, synchronization, and (de)marshaling mechanisms. Alleviating unused object-oriented framework generality (challenge 1) can specialize the middleware for different product variants. Avoiding redundant request creation (challenge 2) occurs in middleware implementations that provide notion of a request message, including CORBA, .NET, and Web Services. Optimizing repeated resolution of same the dispatch (challenge 3) can benefit middleware implementations (such as CORBA, COM, and EJB) that navigate multiple layers/lookup tables to process target requests. Specializing (de)marshaling (challenge 4) and deployment platform generality (challenge 5) can be applied to other middleware that target heterogeneous OS, compiler, and hardware platforms.

3. RESOLVING MIDDLEWARE GENERALITY VIA CONTEXT-SPECIFIC SPECIALIZATIONS

This section examines techniques that focus on the use of context-specific specializations to enhance the QoS of PLA-based DRE systems by alleviating excessive generality in middleware implementations. Context-specific specialization techniques are related to *partial evaluation*, which creates a specialized version of a general program that is more optimized for time and/or space than the original [12]. Context-specific specializations can be realized using *code-refactoring and weaving* [32, 6], which uses aspect-oriented programming mechanisms to factor out and weave cross-cutting concerns, as well as *language mechanisms*, such as program optimization techniques [11]. Below we describe how we applied context-specific specializations to TAO to resolve the challenges described in Section 2. The specialization techniques and tools are reusable and applicable to various middleware implementations beyond TAO.

3.1 Applying Context-Specific Specializations to Middleware

Context-specific specializations described in this paper include constant propagation, layer-folding, memoization, code-refactoring, and aspect weaving. These specialization are driven by *invariant properties* [14], which are specific application-, middleware-, and platform-level characteristics that remain fixed during system execution, but which may vary for different system configurations/-requirements. The invariants themselves may be specific for a particular PLA or applicable to many PLAs. Invariant properties covered in this paper include particular attribute settings (such as timer rates), parameter values (such as arguments to a method), and internal/external contexts (such as a dispatcher for a request and hardware, OS and compiler settings).

In simple cases, an invariant property manifests itself in the form of a call to method `m()`, where one or more of the parameters of the method is always bound to the same value. Our program specialization strategies push invariant data through the middleware code, simplifying along the way. For example, we create a specialized version of `m()` where the parameters with fixed values are removed and the body of `m()` is simplified according to the information provided by the fixed parameter values. Below, we describe the *intent* (purpose), *invariance assumptions* (conditions in our *BasicSP* case study that enabled certain specializations), and *type* (technique) of specialization we applied to resolve the middleware generality challenges described in Section 2.

To evaluate our middleware specializations in a realistic context, we applied them to the TAO implementation of Real-time CORBA, which is written in C++ and contains many general-purpose ORB optimizations [21, 23]. We use general-purpose optimized TAO as a baseline to quantify the benefits of specializations that help resolve the challenges for PLAs described in Section 2.2. We focus on TAO since it is a mature, efficient, and open-source implementation of the Real-time CORBA standard that is used widely in production DRE systems (`www.dre.vanderbilt.edu/users.html`).

3.1.1 Specialize Middleware Framework Extensibility via Aspect Weaving

We first describe specialization techniques for resolving challenge 1 in Section 2.2.

Intent. Eliminate unnecessary extensibility mechanisms (such as indirections and dynamic dispatching) in object-oriented frameworks along the critical request/response processing path. This specialization can be applied to many internal ORB frameworks that handle transport protocols, request demultiplexing, and concurrency models. For our case study, we choose to specialize TAO's (1) Reactor framework [27], which is responsible for demultiplexing connection and data events to their corresponding GIOP event handlers, and (2) pluggable protocol framework [16] which allows ORBs to communicate transparently via different protocol implementations, such as TCP/IP, VME, SSL, SCTP, UNIX-domain sockets, and/or shared memory.

Invariance assumptions. After a Reactor framework implementation is selected for the *BasicSP* scenario, it does not change during the lifetime of the ORB. Likewise, after a protocol implementation is selected it also does not change during the lifetime of the ORB.

Specialization. Figure 3(A) shows different Reactor implementations supported by TAO. The `Select_Reactor` uses the single-

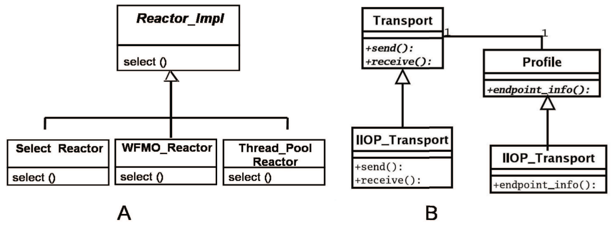

Figure 3: Reactor & Protocol Specialization

threaded `select()`-based event demuxer, the `Thread_Pool_Reactor` uses the multi-threaded `select ()`-based event demuxer, and the `WFMO_Reactor` uses the Windows `WaitForMultipleObjects()` event demuxer. To work transparently across all Reactor framework implementations, TAO uses an abstract base class (*i.e.*, a generic `Reactor_Impl`) that delegates to concrete subclasses via virtual method calls. Specializing the Reactor framework with a concrete subclass (*i.e.*, a subclass with no virtual methods) eliminates the indirection (generality) by using the concrete reactor instance directly.

TAO's pluggable protocol framework uses the Template Method pattern [7] to configure different protocol implementations during the ORB initialization phase. As shown in Figure 3(B), this framework consists of protocol-independent components, such as the `Transport` class that provides `send()` and `recv()` hooks to encapsulate a connection and provide a protocol-independent means of sending/receiving data. Protocol-specific classes, such as the `IIOP_Transport` class, override these hooks to implement protocol-specific functionality. The `Transport` class interacts with other framework components, such as the `Profile` class that encapsulates addressing information in TAO, which in turn uses the Template Method pattern to support multiple protocol implementations. Specializing the hook methods in a template method with protocol-specific behavior eliminates the indirection (*i.e.*, the *virtual* hook methods).

The specializations described above are an example of *aspect weaving*, where the generality (*i.e.*, virtual methods and indirections) that crosscuts different classes and files is customized for a specific context. For example, our *BasicSP* PLA scenario only uses the `Select_Reactor` and VME protocol, so there is no need to incur additional indirection and generality overhead.

3.1.2 *Specialize Request Creation/Initialization via Memoization*

We now describe specialization techniques for resolving challenge 2 described in Section 2.2.

Intent. Rather than creating a new CORBA request repeatedly for each invocation, create/initialize a request once and only update its state that changes.

Invariance assumption. Many (often most) operation parameters and/or context information in a request do not change across invocations in DRE systems.

Specialization. Figure 4 shows the structure of a two-way CORBA request using GIOP version 1.2. As shown in the figure, every

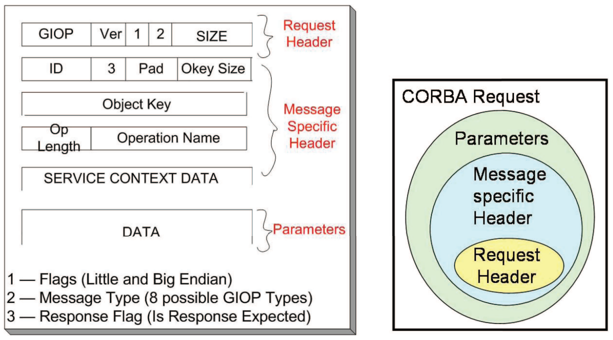

Figure 4: Opportunities for Request Creation Specialization

request has three components defined by the CORBA specification: (1) a request header that indicates the type of CORBA request (*i.e.*, GIOP version 1.0, 1.1, or 1.2) and the total size of the message, (2) a request-specific header containing the object key that uniquely identifies the servant and service context information containing service-specific information, such as the required priority and transaction/security contexts, and (3) optional parameters that were passed as arguments to the operation.

Figure 4 also shows three types of specializations of *increasing strength* that can be applied. In some situations only the request header can be specialized, *i.e.*, its contents are held constant, updating only the total size of the message. In other situations, both the request and the request-specific headers can be held constant, updating only the payload. Finally, the entire CORBA request can sometimes be reused wholesale across multiple request invocations.

This specialization is an example of *memoization*, where a result is precomputed and saved rather than recomputed each time. In our *BasicSP* PLA case study, the precomputed "result" is the CORBA request. This specialization thus avoids unnecessary creation and/or initialization of requests.

3.1.3 *Specialize Dispatch Resolution via Layer-Folding*

We now describe specialization techniques for resolving challenge 3 described in Section 2.2.

Intent. Resolve the target request dispatcher once for the first request and reuse it to service all other requests sent over the same dedicated connection.

Invariance assumptions. The same operation or operations in the same IDL interface are invoked on a multiplexed connection.

Specialization. Figure 5 shows a normal layered demultiplexing path through a CORBA server, *i.e.*, the ORB core locates the target POA, which locates the servant, which locates the skeleton, which dispatches the request to an application-defined method. Rather

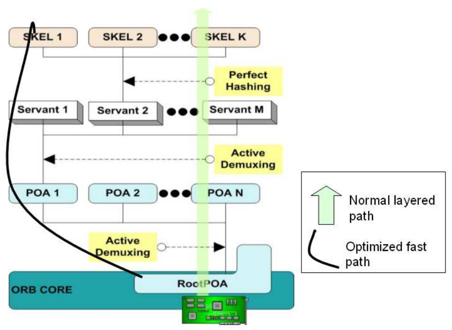

Figure 5: Specializing Request Dispatching

than navigating this layered path, a specialized implementation can cache the skeleton servicing the request and invoke the method on the skeleton directly. A similar approach can be applied to cache the target POA(s) and servant.

This specialization is an example of *layer-folding* plus memoization, where an answer (in our case the dispatcher) is saved for later use than recomputing it each time. This enables multiple middleware layers to be collapsed during request processing.

3.1.4 *Specialize Request Demarshaling via Constant Propagation*

We now describe specialization techniques for resolving challenge 4 described in Section 2.2.

Intent. Eliminate redundant tests for byte order when demarshaling a CORBA request and do not align the individual fields within the request.

Invariance assumptions. The communicating entities reside on nodes with the same byte order, compiler padding/alignment rules, and the same (de)marshaling mechanisms for client(s) and server(s).

Specialization. Standard-compliant CORBA ORBs are required to test for byte order compatibility for every part of a CORBA request (not just the payload), including all fields in the CORBA request and request-specific headers. Figure 4 shows the different parts of a CORBA request. For a typical request with a few basic types (such as `long`, `short`, and `octet` parameters), these tests translate to ∼15–20 byte order tests per request. Removing these redundant tests on homogeneous compiler/middleware platforms can significantly improve demarshaling efficiency, particularly as the data type complexity increases. Similarly, while marshaling a CORBA request, ORB implementations align the individual components, *e.g.*, request size, id and objectkeys, to their natural boundaries. For a typical request with basic types, all the ∼15–20 components must be aligned. Ignoring alignment can improve marshaling efficiency and eliminate padding, thereby reducing request size.

These specializations are an example of *constant propagation*, where the byte-order is propagated along with the request to the recipient and checked to ensure the validity of the invariance assumption. Similarly, unaligned data is sent along with the request to the recipient, where demarshaling fails if data should be aligned.

3.1.5 Specialize Platform Generality via Autoconf Mechanisms

We now describe specialization techniques for resolving challenge 5 described in Section 2.2.

Intent. Choose the right hardware, OS, and compiler settings to maximize application QoS without affecting portability, interoperability, or correctness.

Invariance assumptions. The deployment platform that hosts the product variant remains fixed during the system's lifetime.

Specialization. We use GNU `autoconf` (`www.gnu.org/software/autoconf`) toolkit to apply platform-specific specialization techniques, including:

- **Exception support.** For certain DRE systems, the use of native exception support is unavailable (*e.g.*, not supported by older C++ compilers) or undesirable (*e.g.*, incurs excessive time/space overhead). Certain middleware solutions support platforms that lack exceptions, *e.g.*, CORBA can emulate exceptions by adding an `Environment` parameter at the end of each operation signature. We extended TAO to use GNU `autoconf` to automatically emulate exceptions when compilers lack such capabilities, when users explicitly select this configuration, or when performance tests indicate that emulated exceptions are more efficient than native exceptions.
- **Loop unrolling.** Middleware implementations need to copy data between kernel and middleware and application buffers. An optimization applicable to certain OS/compiler platforms is to unroll the loop of `memcpy()` standard library function up to a certain buffer size. We extended TAO to use GNU `autoconf` to configure middleware automatically to use the either the optimized or default version of `memcpy()`, depending on performance tests that deem one more efficient than the other.

For both specializations, we use GNU `autoconf` to perform these performance tests automatically just before the ORB compilation process begins. Based on the test results, GNU `autoconf` sets certain macros in the TAO source code, which are then used to select which specializations to apply.

3.2 A Toolkit for Automating Context-Specific Specializations

Large-scale DRE systems, such as Boeing's Bold Stroke PLA, contain millions of lines of source code. Implementing specialization techniques by manually handcrafting the optimizations described in Section 3.1 into such large code bases will clearly not scale. To augment GNU `autoconf`, therefore, we have created a domain-specific language (DSL) and associated tools that help simplify two steps in the specialization process: (1) identifying specialization points and transformations and (2) automating the delivery of the specializations.

The remainder of this section describes the *Feature Oriented Customizer* (FOCUS), which is an open-source DSL-based toolsuite and process we developed to automate the specialization of middleware for PLA-based DRE systems. FOCUS is available at `www.dre.vanderbilt.edu/˜arvindk/FOCUS`.

3.2.1 FOCUS Requirements and Goals

Our primary goal for FOCUS was to build a general-purpose DSL, supporting tools, and a process to automate context-specific middleware specializations and then to validate our approach by applying it to TAO. The types of specializations discussed in Section 3.1 yielded the following requirements for FOCUS:

1. Ability to manipulate code. Applying aspect weaving to framework specialization requires the ability to manipulate code, such as performing search and replace specializations to devirtualize hook methods in the `Reactor_Impl` base class and replace them with the name of concrete reactor implementation.

2. Ability to refactor code regions. Object-oriented framework specializations need to move specialized code (*e.g.*, concrete implementations of hook methods) from a derived class to the new concrete class. Similarly, additional header files and methods may may need to be moved from/to a derived class to/from a new concrete class. Likewise, layer-folding optimizations require the capability to inject code that bypasses layers at specific locations in the code.

3. Ability to remove code. Code refactoring, memoization, and aspect weaving specializations require the removal of certain redundant functionality. In memoization optimizations, for instance, redundant functionality that repeatedly creates the same request must be replaced with code that caches the request header.

To support these requirements, FOCUS uses generative programming techniques [3] that combine annotations (directives) with code templates [25] to specialize middleware implementations. FOCUS annotations are embedded within the middleware source to identify points of variability, *e.g.*, where a dispatching decision is made or a particular protocol is created. This approach enables most of the basic infrastructure of the middleware to remain fixed, but identifies well-defined variability points where specializations can be woven automatically. It also enables middleware developers to explicitly know the variability points when source code changes are made, thereby minimizing the skew between specializations and an evolving middleware source base.

3.2.2 Automating Middleware Specializations with FOCUS

The process of applying FOCUS DSL and tools can be executed in three phases by middleware developers and application PLA developers, as discussed below.

Phase 1: Capturing specialization transformations. In this phase, middleware developers capture the code-level transformations required to implement a specialization using the *FOCUS Specialization Language* (FSL), which is a DSL that supports the following types of specializations: (1) search and replace transformations (<search> ... <replace>), (2) copying text from different positions in multiple files onto a destination file (<copy-from-source>), (3) commenting regions of a program (<comment>), and (4) removing text from a program (<remove>). FSL uses an XML DTD to capture the transformations, which facilitates ease of (1) *extension*, *i.e.*, additional transformations can be represented via new XML tags and (2) *transformation*, *i.e.*, XSLT transformations can be used to transform the specializations onto different tool input formats. Similar approaches have been used in commercial tools, such as Ant (`ant.apache.org`), which use XML to capture build steps and rules.

The output of phase 1 is a set of FSL specialization files that capture all the transformations needed for the specializations. Section *4.2.1* illustrates portions of the transformations required to automate aspect weaving specializations in TAO.

Phase 2: Middleware annotation. In this phase, middleware developers use the FSL specialization files to annotate the middleware with metadata required for the desired transformations. Annotations in FOCUS are only required for transformations that copy, comment, or add code, *i.e.*, when using the <comment>, <copy-from-source>, or <add> tags, respectively. Other transformations, such as search and replace, and remove do not require annotation. Metadata is inserted as special comments in the source code using source language syntax for comments. This metadata is used by FOCUS to aid in the transformation of source code, but is opaque to compilers for general-purpose languages like C++ or Java and imposes no extra overhead on general-purpose middleware source code.

During middleware evolution, such as feature addition/modification, middleware developers must respect the annotations. For example, any new code between annotations that mark the begin and end of a copy/comment does not require changes to the specialization files. Section *4.2.1* illustrates how annotations along with <copy-from-source> tags can be used to minimize skew between specializations and middleware source code.

Annotations help the FOCUS transformation process by enabling a lightweight specialization approach that does not require a full-fledged language front-end to parse the entire source code to identify the specialization points (hooks). This approach enables FOCUS to work across middleware implementations in different languages, *e.g.*, hooks can be left within a C++- or Java-based middleware implementation for FOCUS to weave in code. FOCUS ascribes no significance to the names for hooks, *i.e.*, they can be arbitrary as long as there is a corresponding name in the specialization file.

Phase 3: Executing specialization transformations. In this phase, PLA developers perform the steps shown in Figure 6. First, they

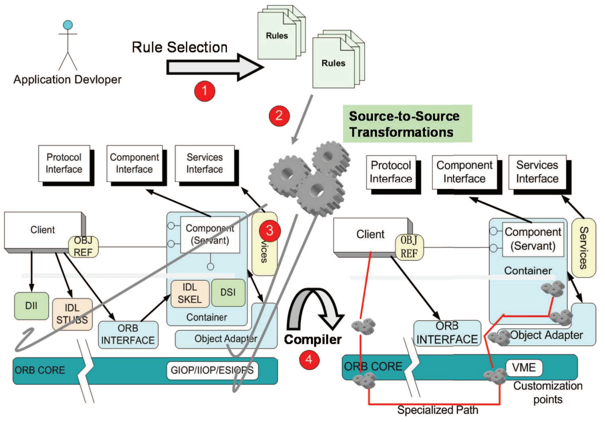

Figure 6: Steps in the FOCUS Transformation Process

select the specializations suitable for the variant (step 1) during the offline SCV analysis phase. Based on the features selected for specialization, the FOCUS transformation engine queries the specialization repository to select the right file(s) (step 2). Based on the transformation rules in the specialization file, the transformation engine executes the transformations (step 3). A compiler for the general-purpose language used to write the middleware is then used to generate executable platform code from the modified source file(s) (step 4).

The FOCUS transformation engine is written in Perl to leverage its mature regular expressions support. Regular expressions enhance the richness of the transformations that can be specified within FSL specialization files. For example, search and replace capabilities in FOCUS leverage regular expressions to ignore leading trailing white spaces and newline characters.

3.3 Summary

This section described the specialization techniques, language, and tools we developed to resolve middleware generality challenges in Section 2. Table 1 lists the specialization techniques along with the corresponding FSL features we applied to resolve the generality challenges (Section 4.2 describes why we did not use FOCUS to resolve (de)marshaling checks and deployment generality).

Similar to the challenges, the FOCUS specialization techniques

Specialization	**Technique**	**FSL features**
Request creation	Memoization	search, replace, add
Demarshaling checks	Constant propagation	Not Applicable
Dispatch resolution	Memoization + layer-folding	search, replace, add
Framework generality	Aspect weaving	add, copy-from-source search, replace
Deployment generality	autoconf	Not Applicable

Table 1: Summary of Specialization Techniques

we describe are reusable and applicable to middleware that incur generality challenges. These techniques modify only copies of the object-oriented framework and middleware code and do not necessitate any modifications to application code or original frameworks and middleware.

4. APPLYING OUR SPECIALIZATION TOOLS TO TAO FOR THE BASICSP SCENARIO

This section presents the results of applying our specialization tools described in Section 3 to a TAO-based implementation of the *BasicSP* scenario. These results help in quantitatively and qualitatively evaluating the extent to which specializations improve the throughput, average- and worst-case latency, and jitter of standard-based middleware implementations. The constant propagation and code refactoring techniques described in the paper were automated using GNU `autoconf` conditional compilation techniques described in Section *3.1.5*. The memoization, layer-folding, and aspect weaving were automated via the FOCUS toolkit described in Section *3.2.2*.

4.1 Analyzing General-purpose Middleware

To specialize general-purpose Real-time CORBA middleware for PLA-based DRE systems, we first analyzed the end-to-end critical code path of the following synchronous two-way CORBA operation in TAO:

```
result = object→operation (arg1, arg2)
```

A path represents a segment of the overall end-to-end flow through the system. The *critical path* is the sequence of steps that are always necessary in TAO to process events, requests, or responses for synchronous and asynchronous operation invocations. This code path is the same for processing the `get_data()` two-way operation and `data_avail` and `timeout` events in the *BasicSP* scenario described in Section 2.1. We use this code path to provide a baseline for comparing our context-specific specializations to quantify the number of steps specialized along the critical request/response processing path shown by the numbered bullets in Figure 7.

Using this figure as a guide, we now describe the steps involved

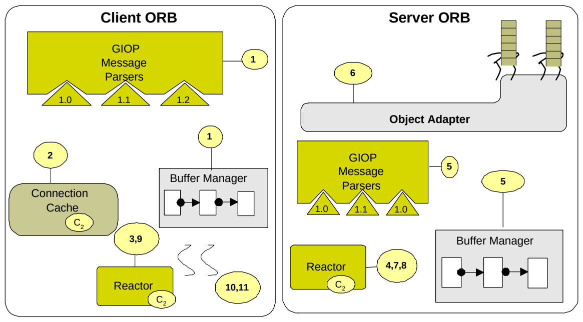

Figure 7: End-to-End Request Processing Path

when a client invokes a synchronous two-way operation. After the connection from client to server process has been established, the following activities are performed by the TAO client ORB when a client application thread invokes an operation on an object reference that designates a particular target object on a TAO server ORB:[2]

1. `Buffer_Manager` allocates a buffer from a memory pool. The GIOP message parser is used to marshal the parameters in the operation invocation.
2. Send the marshaled data to the server using the established connection *e.g.*, C_2.
3. The leader thread waits on the `Reactor` for a reply from the server. The follower thread waits on a condition variable or a semaphore.

The server ORB activities for processing a request are described below:

4. Read the header of the request arriving on connection C_2 to determine the size of the request.
5. `Buffer_Manager` allocate a buffer from a memory pool to hold the request and an appropriate GIOP message parser is used to read the request data into the buffer.
6. Demultiplex the request to locate the target portable object adapter (POA) [22], servant, and skeleton – then dispatch the designated upcall to the servant after demarshaling the request parameters.
7. Send the reply (if any) to the client on connection C_2.
8. Wait in the reactor's event loop for new connection and data events.

Finally, the client ORB performs the following activities to process a reply from the server:

9. The leader thread reads the reply from the server on connection C_2.
10. The leader thread hands off the reply to the follower thread by signaling the condition variable used by the follower thread.
11. The follower thread demarshals the parameters and returns control to the client application, which processes the reply.

4.2 Specializing TAO Middleware

Having outlined the activities at the client and server, we now (1) describe how we specialized TAO using invariance assumptions to resolve the challenges for PLAs described in Section 2.2 in the context of the Bold Stroke *BasicSP* scenario and (2) quantitatively compare the end-to-end latency, throughput, and predictability improvements accrued from our approach. All experiments were performed on an Intel Pentium III 851 Mhz processor with 512 MB of main memory running on Linux 2.4.7-timesys-3.1.214 kernel, which contains a very predictable real-time kernel module. The TAO middleware used for the experiments was version 1.4.7, which was compiled with gcc version 3.2.2.

To ensure portability and interoperability, our specializations largely comply with the Real-time CORBA specification and do not modify any standard interfaces or *BasicSP* application code. Our specializations resolve middleware generality challenges illustrated in Figure 2, are applied along the critical request/response processing path and affect end-to-end QoS. Our approach specializes most of the steps (1, 3, 11 on the client and 4, 6, 8 on the server).

TAO provides a wide range of configuration options (see `www.dre.vanderbilt.edu/~schmidt/TAO-options.html`). For this analysis, we used the following configuration: (1) portable interceptors are not used, (2) servants inherit statically from `org::omg::PortableServer::Servant`, *i.e.*, we do not consider CORBA's dynamic invocation/skeleton features, (3) no proprietary policies were used in the ORB, and (4) TAO's general-purpose optimizations (*e.g.*, active demultiplexing, perfect hashing, and buffer caching strategies) were enabled for all experiments. These assumptions are representative of ways in which DRE systems commonly apply Real-time CORBA middleware [26, 31].

To showcase our results, a sample size of 100,000 data points was used to generate results from following types of experiments for each specialization presented in Sections *3.1.1* through *3.1.5*:

1. **End-to-end latency metrics**, which measure the differences in end-to-end latency/throughput between general-purpose and specialized versions of TAO. For each experiment, high-resolution timers on the client collected end-to-end measurement data used for analysis.
2. **Path specialization metrics**, which compare latency measures for specialized vs. general-purpose critical paths. For each experiment, high-resolution timers within TAO's ORB core measured latency improvements for the code path specialized within TAO.
3. **Cumulative metrics**, which measure the end-to-end latency and predictability improvements accrued by applying all specializations.

For each specialization, we describe (1) the steps specialized along the request/response processing path, (2) how the specialization was automated, and (3) consequences of applying our specialization in terms of CORBA compliance and applicability. For our experiments, we define *predictability* as the measure of standard deviation of the data points.

4.2.1 Applying the Aspect Weaving Specialization

This specialization corresponds to step 3 and 9 in the client side and step 8 in the server side of Figure 7.

Specialization automation. Specializing the Reactor component involved (1) replacing the `ACE_Reactor_Impl` class with the concrete `ACE_Select_Reactor` implementation within the reactor, (2) replacing the creation of other reactors with the specialized version in ORB factory methods [7], and (3) eliminating virtual methods from the reactor and interfaces within the middleware. To automate the specialization, we used FSL to capture the transformations, some of which are shown in Listing 1. Lines 1–2 capture the module (directory or package) and file on which the transformations are done. Devirtualizing interfaces of the reactor is captured in line 3. Lines 4–8 allow replacing the

[2]This discussion has been generalized using the Reactor, Acceptor-Connector, and Leader/Followers patterns [29], which are used in many popular CORBA ORBs, such as e*ORB, ORBacus, Orbix, and TAO.

```
1: <module name="ace">
2:  <file name="Reactor.h">
3:    <remove>virtual</remove>
4:    <substitute>
5:      <search>ACE_Reactor_Impl</search>
6:      <replace>ACE_Select_Reactor</replace>
7:    </substitute>
8:  </file>
9: </module>
10: <module="TAO/tao">
11:  <file name="advanced_resource.cpp">
12:   <comment>
13:    <start-hook>TAO_REACTOR_SPL_COMMENT_HOOK_START</start-hook>
14:    <end-hook>TAO_REACTOR_SPL_COMMENT_HOOK_END</end-hook>
15:   </comment>
16:  </file>
17: </module>
```

FOCUS Listing 1: Reactor Specialization

`ACE_Reactor_Impl` with the desired concrete select reactor. Similarly, lines 12–15 show how unspecialized code within two points in the file (<start-hook> ... <end-hook>) is commented out for the transformations.

Listing 2 shows how we annotate the middleware source code with hooks based on the FSL specialization file (Listing 1).

```
//File: advanced_resource.cpp
ACE_Reactor_Impl*
TAO_Default_Resource_Factory::
  allocate_reactor_impl (void) const
{
  ACE_Reactor_Impl *impl = 0;
  /* FOCUS: Comment hook */
//@@ TAO_REACTOR_SPL_COMMENT_HOOK_START
  ACE_NEW_RETURN (impl, ACE_TP_Reactor, 0);
//@@ TAO_REACTOR_SPL_COMMENT_HOOK_END
  return impl;
}
```

FOCUS Listing 2: Annotated Middleware Source Code

Listing 3 illustrates how FOCUS transformed source code so that the base Reactor (`ACE_Reactor_Impl`) is replaced with the specialized Reactor (`ACE_Select_Reactor`). This specialization is validated by our invariance assumption that after `ACE_Select_Reactor` is selected, it does not change for the *BasicSP* scenario. Another observation is that the annotations are preserved during the transformation process, which enables multiple specializations to use the same hook for specifying transformations, thus avoiding cluttering of hooks within the middleware source code.

To specialize TAO's pluggable protocol implementation, we used <copy-from-source> capabilities provided by FSL. Listing 4 shows

```
//File: Reactor.h
class Reactor
{
public:
  int
   run_reactor_event_loop (REACTOR_EVENT_HOOK = 0);
  // Other public methods ....
private:
// Code woven by FOCUS:
  ACE_Select_Reactor *reactor_impl_;
// End Code woven by FOCUS
};

// File: advanced_resource.cpp
// Code woven by FOCUS:
ACE_Select_Reactor*
// End Code woven by FOCUS
TAO_Default_Resource_Factory::
  allocate_reactor_impl (void) const
{
// Code woven by FOCUS:
  ACE_Select_Reactor *impl = 0;
// End Code woven by FOCUS
  /* FOCUS: Comment hook */
//@@ TAO_REACTOR_SPL_COMMENT_HOOK_START
//  ACE_NEW_RETURN (impl, ACE_TP_Reactor, 0);
//@@ TAO_REACTOR_SPL_COMMENT_HOOK_END
// Code woven by FOCUS:
}
```

FOCUS Listing 3: Transformed Middleware Source Code

how the concrete protocol specific implementations of template methods defined in the `Profile` class are copied from the `IIOP_Profile` class. As shown in the listing, <copy-hook-start> <copy-hook-end> tags signify the start and end locations of the template method implementations in the `IIOP_Profile` class. These concrete implementations are copied on to the base `Profile` class at a location defined within the `Profile.cpp` file. The advantage of this design is that changes made to the implementations of the template methods in `IIOP_Profile.cpp` do not necessitate a change to the specialization file. In fact, after we completed this specialization, support of IPv6 protocol was added to TAO, but our specializations required no changes.

```
<file name="Profile.cpp">
 <copy-from-source>
  <source>IIOP_Profile.cpp</source>
   <copy-hook-start>PROFILE_METHODS_COPY_HOOK</copy-hook-start>
   <copy-hook-end>PROFILE_METHODS_COPY_HOOK_END</copy-hook-end>
   <dest-hook>PROFILE_SPL_ADD_HOOK</dest-hook>
 </copy-from-source>
</file>
```

FOCUS Listing 4: Protocol Specialization

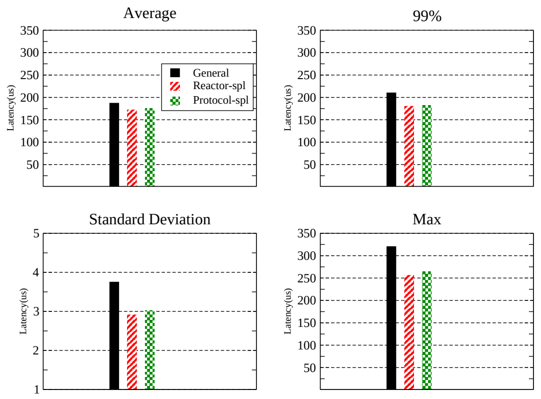

Figure 8: Results for Reactor & Protocol Specializations

Empirical results. Figure 8 illustrates the improvements to end-to-end latency by specializing two object-oriented frameworks used in TAO. Since reactors and protocols are used by both client and server ORB, we present the most representative end-to-end results. Our specialization results in average latency improvements of ∼8μsecs (4%) for the reactor case and in ∼10μsec (5%) for the protocol case. These specializations also minimize dispersion measures for both the specializations, though not appreciably. The 99% and worst-case measure also decrease, thereby showing how removing virtual method indirection enhances predictability.

Our results show how minimizing dynamic dispatch along the critical path can improve performance. Similar approaches can be applied to specialize other frameworks used within middleware. Based on the results obtained in this case study, our future work will specialize other object-oriented frameworks in TAO to further improve its performance.

Applicability and CORBA compliance. Specialization of object-oriented framework extensibility can be applied to all ORB implementations that use virtual methods, yet can be customized via a single concrete instance late in the system lifecycle, *e.g.*, during deployment or initialization. This specialization is CORBA-compliant since the reactor is part of TAO's ORB core implementation, not part of the public API defined by the Real-time CORBA specification. Similarly the specialization of the protocol framework does not modify any standard APIs or application code, but only affects hook methods specific to TAO's implementation.

4.2.2 Applying the Memoization Specialization

This specialization corresponds to step 1 in the client side of Figure 7.

Specialization automation. In TAO, the GIOP engine creates protocol specific request/response objects. Listing 5 illustrates a portion of the transformations for this specialization. During the first invocation of a request/response, the length of the actual header is computed and cached (as shown in lines 1–6). For subsequent requests the cached pre-marshaled header is used by moving the current writable location by the total header size.

```
<add>
 <hook>TAO_HEADER_CACHING_ADD_HOOK</hook>
 <data>
1.   if (__header_cached__)
2.   {
3.     // First invocation -- normal path
4.     __header_cached__ = 0;
5.     this->write_header (...);
6.     skip_length__ = this->total_length ();
7.   }
8.   else
9.   {
10.     // All invocations -- Optimized path
11.   this->skip (skip_length)
12.   }
 </data>
</add>
```

FOCUS Listing 5: Specializing Request Creation

We applied specialized request creation to TAO on a per-connection basis, *i.e.*, the request headers cached are specific to a connection. This design stems from our invariance assumption from the *BasicSP* scenario, where the `get_data` and `data_avail` operation are sent along separate connections.

Empirical results. Figure 9 illustrates the end-to-end and code path specialization improvements that result from applying the request creation/initialization specialization on the request and request-specific CORBA header. The average end-to-end latency measures improved by ∼8μsec (4%), while the path specialization results improved by 25%. This discrepancy shows how much our specialized code path influences end-to-end latency. The dispersion measures improve, but not significantly, by applying this specialization. Both 99% and worst-case measures improve, which show this specialization improves predictability. These results show how the end-to-end path specialization results are influenced by the contribution from the actual path specialized.

Applicability and CORBA compliance. Specializing the entire request is possible only if the request does not change, which is the case for control messages sent between `Timer` and `GPS` components. Specializing the request and request-specific header is possible if only the contents change between requests, which is the case for the `get_data()` operation. This specialization can be applied for the standard Real-time CORBA SERVER_DECLARED priority model, where the priority information is set *a priori* during object reference creation. Specializing only the request header is applicable to all requests, though it has the least payoff in terms of improvements in performance since it represents a relatively small portion of the request. All three approaches comply with the CORBA specification since they do not change the type of the CORBA request message. The third approach, however, does not update the request identifier, which is used to uniquely identify the client thread processing the response when multiplexed connections are used.

4.2.3 Applying the Layer-Folding Specialization

This specialization corresponds to step 6 to step 8 in the server side of Figure 7.

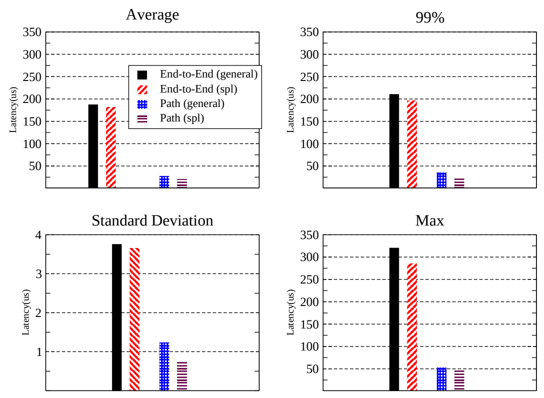

Figure 9: Results for Request Creation/Initialization Specialization

Specialization automation. We implemented the layer-folding specialization (Section 3.1) by caching the target request dispatcher determined when the first request from the client on a connection is serviced. Subsequent requests used the cached dispatcher directly, *i.e.*, the skeleton that services the requests. FSL annotations were add to TAO's POA so FOCUS can weave in code that cached the skeleton servicing the request. Another annotation within the ORB core marked the start of the normal request path.

These specialization transformations were similar to the aspect weaving and memoization specializations discussed in Section 3.1 and are applied on a per-connection basis. Multiple simultaneous client connections that have different request dispatcher can therefore be serviced concurrently. This design conforms to our invariance assumption from the *BasicSP* scenario that the operations are same only on a per-connection basis.

Empirical results. Figure 10 illustrates the end-to-end and code path performance improvements resulting from the dispatch resolution specialization. The average end-to-end latency measures improved by ∼30μsecs, which is ∼16% better than the general-purpose TAO implementation. For the actual code path specialized this translates to ∼40% improvement in latency. The dispersion measures for end-to-end latencies improved by a factor of ∼1.5, while those for the specialized path were twice as good as those for the general-purpose path. The 99% measures are similar to the dispersion measures, indicating improvement in predictability. The worst-case measures improved by 20% when applying the dispatch resolution specialization to the specialized path and by ∼14% for the end-to-end results. These results show that applying layer-folding specialization to the TAO middleware improves predictability and latency considerably.

Applicability and CORBA compliance. This specialization applies to the `get_data()` operation in the *BasicSP* scenario where the same operation is invoked repeatedly. Caching the target servant and skeleton sacrifices some CORBA compliance since thread-specific state (*e.g.*, CORBA Current and POA Current are not maintained. This context information is often unnecessary, however, *e.g.*, the POA current interface is used primarily when the POA is associated with a `Default_Servant` (where one servant handles all invocations) or `Servant_Manager` (which creates a servant dynamically to handle requests). Since these dynamic CORBA policies are rarely – if ever – used in DRE systems, the impact on CORBA compliance is negligible *in this context*.

4.2.4 Applying the Constant Propagation Specialization

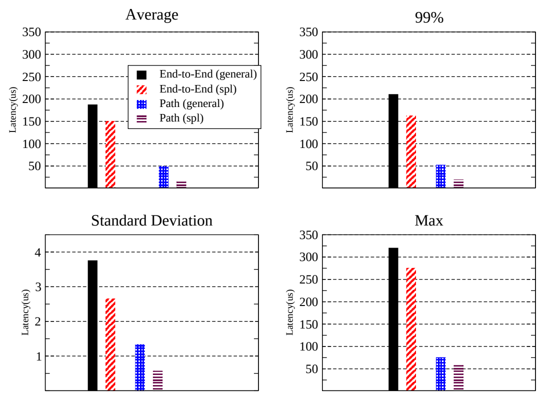

Figure 10: Results for Dispatch Resolution Specialization

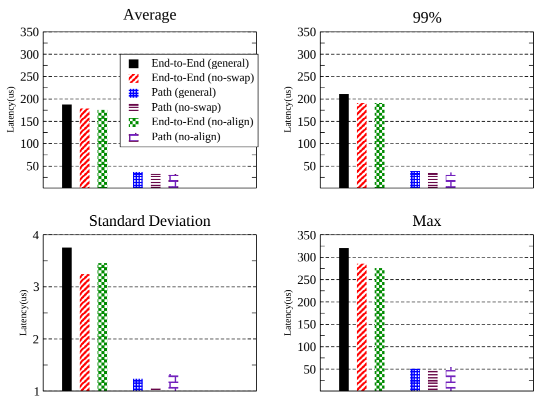

Figure 11: Results for Request (De)marshaling Specialization

This specialization corresponds to steps 1 and 11 on the client and steps 4 and 6 in the server of Figure 7.

Specialization automation. The (de)marshaling engine within TAO used several OS platform-specific capabilities that were automatically (un)set using GNU `autoconf`. GNU `autoconf` necessitated the use of conditional compilation to implement this specialization. Two flags, `CDR_IGNORE_ALIGNMENT` and `DISABLE_SWAP_ON_READ` were added to the `write()` and `read()` methods within TAO's Common Data Representation (CDR) engine to ignore alignment and byte-order values in the request/response fields. This design conforms to our invariance assumption that the communicating entities run on homogeneous middleware, OS, compiler, and hardware platforms, which is often the case for production DRE systems.

Empirical results. Figure 11 illustrates the end-to-end and path performance improvements from applying the request (de)marshaling specialization. The specialized path for this experiment began when a server demarshaled a request until the response was returned to the client. Applying the specialization that ignored alignment improved end-to-end latencies by ∼8μsecs (a 4% improvement over the general-purpose TAO implementation), while eliminating byte order checks improved byte order checks by ∼9μsec (a 4% improvement over the general-purpose TAO implementation). Path specialization results improved by ∼ 4 – 5μsec (a 10% improvement) for both the cases. Although the general-purpose TAO implementation performs tests on the client and server for all fields in a CORBA request header, our specialization improvements were relatively small since our initial experiment sent a single `long` data type, which required very few byte order tests.

To quantify the improvements from this specialization for more complex data types, we conducted another experiment that sent an IDL structure with four primitive types, a `short`, `long`, `double` and `float` interspersed with a `char`. The use of a `char` type forced the general-purpose TAO middleware to re-align the individual primitive types. The specialized TAO middleware, however, did not incur this overhead.

A `sequence` of this structure with varying sizes was sent over the network to measure the improvement in performance. Both specializations were enabled simultaneously for this experiment. Table 2 illustrates the speed up in average end-to-end latency accrued from applying our specialization. The results show that latency measures improve between 12 – 30% with increasing sequence lengths.

These results underscore the fact that the benefits of specializations often depend heavily on the use cases that exercise the specialized code.

Sequence Length	Speedup
64	11.5%
128	17.35%
1,024	20.12%
2,048	25.64%
4,096	30.12%

Table 2: Performance Speedup as a Function of Sequence Length

Applicability and CORBA compliance. Elimination of byte-order checks and ignoring alignment specializations are applicable to deployments on homogeneous environments *i.e.*, nodes with the same byte order, *e.g.*, *NodeA* and *NodeB* in our *BasicSP* scenario, and/or the same platform implementations at sender and receiver. These specializations are not CORBA compatible. A middleware implementation, however, can add recovery mechanisms, such as checking for byte order within the request before using the aforementioned specializations, though these mechanisms violate our invariance assumption.

4.2.5 *Applying Autoconf Techniques for Platform Specialization*

This specialization corresponds to the underlying platform on which the *BasicSP* scenario was run.

Specialization automation. To automate the loop unrolling optimization, we used GNU `autoconf`'s `AC_RUN_IFELSE` capability that compiled and executed a benchmark to compare performance both with and without our optimization. If our optimization was faster, `autoconf` sets the `ACE_HAS_MEMCPY_LOOP_UNROLL` flag to enable the feature. For exception support, we used GNU `autoconf`'s `AC_COMPILE_IFELSE` feature to determine if a compiler supported exceptions and then empirically evaluated whether using native exceptions was faster than emulated exceptions.

Empirical results. Figure 12 illustrates how applying our loop unrolling and exception emulation specialization techniques together improved average end-to-end latency measures by ∼17%. Worst-case latency improved by ∼12%, while the 99% latency measures were closer to the average for our specializations, thereby indicating better predictability. These results show that specializing deployment platforms via GNU `autoconf` can improve QoS significantly.

Applicability and CORBA compliance. The GNU `autoconf` specialization techniques do not affect specification compliance at all.

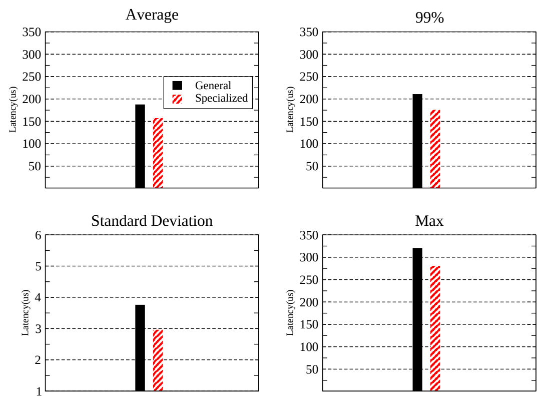

Figure 12: Results for Specializing Deployment Platform

4.2.6 *Applying the Specializations Cumulatively*

Figure 13 illustrates the QoS improvements accrued by applying all of the middleware specializations discussed above to a remote CORBA operation. The average end-to-end latency for the specialized TAO dropped by ∼43%, while the dispersion measure was twice as good as general-purpose optimized TAO implementation, indicating considerable improvement in predictability, which is essential for DRE systems.

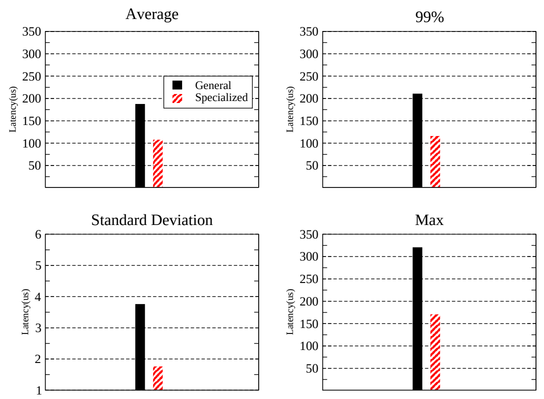

Figure 13: Results for Cumulative Specialization Application

Similarly, the 99% bound values for the specialized TAO improved by ∼40% while worst-case measures improved by ∼150μsecs, which is a 45% improvement over the general-purpose TAO implementation. End-to-end throughput measures improved by an average of ∼65%. To measure performance speed up for a complicated data structure, we ran the experiment using the complex data structure from our demarshaling experiments. For a sequence length of 64 average latency improved by ∼26%, while for a length of 4,096 latency improved by ∼51%.

4.3 Evaluating FOCUS

Now that we have illustrated how FOCUS's DSL, tools, and process can be applied to help middleware developers build and evaluate middleware specializations, we can evaluate its benefits and drawbacks.

Benefits. In resolving the challenges described in Section 2, FOCUS has the following benefits:

- It preserves portability of the middleware implementations it specializes, *i.e.*, the specialized middleware should run on all platforms on which the middleware runs. The FSL snippets in Section 4.2.1 do not change the interface of Reactor or protocol components in TAO.
- It has no external dependencies, *i.e.*, it does not require external libraries to be linked for execution.
- It supports role separation, *i.e.*, middleware developers capture the specialization and annotate the middleware, whereas PLA application developers select the specializations based on SCV analysis.
- It uses COTS tools and standard technologies, such as Perl and XML, to automate the delivery of these specializations to enhance its use in production middleware platforms.
- Its transformations incur no unnecessary overhead at runtime since they are performed statically at compile-time, similar to other source-to-source transformations, such as AspectC++ (`www.aspectc.org/`), DMS (`www.semdesigns.com`), TXL (`www.txl.ca`), and Stratego-XT (`www.program-transformation.org/Stratego/`). The transformed middleware source code woven by FOCUS (Section 4.2.1) illustrates that there is no tool-specific code inserted as a part of the transformation process.
- Its specializations do not affect business logic and only modify the structure of middleware implementations, particularly object-oriented frameworks. The non-transformed versions of the frameworks are therefore still available when other developers need to use their object-oriented extensibility features. None of the specializations described earlier modified or specialized *BasicSP* application code.

Drawbacks. Since FOCUS was developed primarily to help us evaluate the benefits of middleware specializations, in general, and the TAO ORB, in particular, it currently has the following limitations:

- It automates the delivery of specializations, but not the identification of specializations suitable for a PLA or an individual variant.
- Developers are responsible for ensuring that annotations are synchronized with specialization rules, *i.e.*, if the annotations are changed the specialization files also need change. This can be ameliorated somewhat by providing guidelines to middleware developers and enhancing the parser to first make sure the required hooks are present in the middleware before performing the transformations.
- Modifications/enhancements to the state and/or- interface of implementations require manual changes to the specializations, *i.e.*, if the name of an operation or its parameters change, the specialization files need to be updated. This limitation, however, is not specific to FOCUS but also to DMS and AspectC++.
- The FOCUS transformation engine does not check that the woven code executes correctly, which is a common limitation with other source-to-source transformation tools, such as AspectC++, that rely upon general-purpose compilers and automated quality assurance tools to ensure the transformations compile and run properly.

Now that we have validated the benefits of automating middleware specializations, we plan to address these limitations in our future work on more powerful specialization languages.

4.4 Summary

Specialization is a promising technique for alleviating the time/space overhead stemming from excessive generality in standards-based middleware implementations and improving its QoS, such as reducing latency and jitter. This section quantified the benefits of specializations we applied to the TAO Real-time CORBA based on invariants stemming from the *BasicSP* scenario, which itself is based on the SCV analysis embodied in the Boeing Bold Stroke PLA. Our empirical results illustrate the viability of our approach to improve the QoS of PLA-based DRE systems, where stringent performance requirements must be met while also preserving application source code and middleware portability/interoperability as much as possible.

The specialization techniques discussed in this section are targeted to PLA developers who identify the applicability of various specialization techniques to a variant in the PLA during SCV analysis. Among the specializations examined and implemented in this paper, deployment platform-specific time/space specializations yielded the most improvement in QoS, followed by memoization, layer-folding, and aspect weaving, in that order. Our future work we will examine other dimensions of specializations such as eliminating layering at the client side and resolving other platform specific generality.

5. RELATED RESEARCH

This section compares our work on context-specific specializations for middleware-based PLAs for DRE systems with related research in a range of system and application domains.

5.1 Operating systems

Specialization techniques have been applied to operating systems. For example, the Synthesis Kernel [19] generated custom system calls for specific situations to collapse layers and eliminate unnecessary procedure calls. This approach has been extended to use incremental specialization techniques. For example, [18] have identified several invariants for an OS `read()` call on HP/UX. Based on these invariants, code is synthesized to adapt to different situations. Once the invariants fail, either re-plugging code is used to adapt to a different invariant or default unoptimized code is used. Our work extends the range of specializations to encompass middleware invariants in the context of PLA-based DRE systems, which have some different constraints. For example, we do not consider dynamic re-plugging costs since it would unduly increase jitter for product variants in many DRE systems.

5.2 Middleware

Specialization techniques have also been applied to various generations of middleware. [14] describes the use of the Tempo C program partial evaluator tool to automatically optimize common software architecture structures with respect to fixed application contexts. For instance, the authors show how partial evaluation can be applied to fold together and optimize layers in early generations of middleware, *i.e.*, a remote procedure call (RPC) implementation, by specializing RPC invocations to the size and type of remote procedure parameters (yielding speed-ups of 1.7x and 3.5x).

[14] also customize a publish/subscribe framework to a context in which subscribers of a particular event are known *a priori*. This type of architecture is representative of the structures encountered in middleware for PLA-based DRE systems, which motivated us to consider similar techniques that could be applied with good effect to optimize Real-time CORBA implementations. In our work, we have identified additional CORBA architectural structures that are amenable to optimization via specialization. Technical challenges remaining include extending the automatic C program techniques in [14] to richer object-oriented languages, such as C++ and Java, that place a greater emphasis on dynamically created data.

Other research on middleware [32, 8] has applied aspect-oriented programming (AOP) techniques to factor out cross-cutting middleware features, such as portable interceptors, (de)marshaling routines, and dynamic typing. Some specializations described in this paper can be implemented using AOP. The primary difference is that our specializations focus more on the *transformations* (woven code) required to specialize middleware, whereas AOP is more of a delivery mechanism to realize specializations.

5.3 Empirically-guided Optimizers

The ATLAS [13] numerical algebra library uses an empirical optimization engine to set the values of optimization parameters by generating different program versions that are run on various hardware/OS platforms. The output from these runs are used to select parameter values that maximize performance. Similarly, our GNU `autoconf` specializations run empirical benchmarks on the target deployment platform to determine the OS, compiler, and hardware parameters that maximize performance.

6. CONCLUDING REMARKS

This paper describes how specializations can be automated and applied to optimize excessive generality in standards-based middleware implementations used for PLAs. We show how context-specific specializations based on the Bold Stroke avionics mission computing PLA can be used to optimize the TAO Real-time CORBA implementation. Our middleware specialization results collectively improved the throughput of Bold Stroke *BasicSP* scenario by $\sim$65%, its average- and worst-case end-to-end latency measures by $\sim$43% and $\sim$45%, respectively, and its predictability by a factor of two, without affecting portability, standard middleware APIs, or application software implementations, while preserving interoperability wherever possible. These improvements are particularly notable since TAO has already been tuned via many general-purpose middleware optimizations [20, 21, 23]. We also described how GNU `autoconf`, FOCUS DSL and tools were used to automate the middleware specializations described in the paper.

The remainder of this section discusses the consequences and implications of our specialization techniques and tools.

Implications on QoS. The specializations discussed in this paper had no inter-dependencies, *i.e.*, the specializations do not overlap in the end-to-end code path. As middleware and system architects develop a catalog of specializations, it will be necessary to document the interplay between the specializations and analyze the implications on mixing and matching different specializations. Similarly, not all the specializations will be applicable to every PLA application scenario, so PLA developers will need to work in conjunction with middleware developers to determine the applicability of the different specialization techniques to product variants.

Quantitative results show that improvements from applying our specializations can be scenario-specific. For example, the demarshaling results showed how a complicated structure benefited more from the specialization than a simple type. When the specialized path is traversed more often, therefore, its influence on end-to-end performance is more significant.

Implications on adaptability. The specialization mechanisms discussed in this paper do not consider adaptation costs, *i.e.*, the overhead of handling and recovering from situations where the invariance assumptions are violated. Adding such mechanisms require activities (such as loading new libraries or adding run-time checks) that can incur considerable jitter, and thus are not desirable for DRE systems.

Implications on schedulability. In many DRE systems, real-time tasks are scheduled and analyzed offline to ensure they complete before their deadlines. Latency overheads caused by general-purpose middleware implementations may cause deadline misses for critical tasks scheduled a priori. Applying our specializations could reduce middleware overhead considerably, helping ensure that critical tasks complete before their deadlines. Our optimizations might also enable such tasks to finish well ahead of their deadlines, thereby increasing the total *slack*, *i.e.*, time interval available for scheduling other tasks (such as soft real-time tasks), in the system. More available slack could potentially increase the number of schedulable soft real-time tasks in the system.

7. REFERENCES

[1] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[2] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6), November/December 1998.

[3] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[4] G. Daugherty. A proposal for the specialization of ha/dre systems. In *Proceedings of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM 04)*, Verona, Italy, Aug. 2004. ACM.

[5] B. S. Doerr and D. C. Sharp. Freeing Product Line Architectures from Execution Dependencies. In *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[8] C. Z. D. Gao and H.-A. Jacobseon. Towards Just In Time Middleware Architectures. In *Proceedings of the 2005 Aspect Oriented Software Engineering Conference (AOSD)*, Nov 2005.

[9] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, New York, 2004.

[10] J. Hatcliff. An Introduction to Online and Offline Partial Evaluation using a Simple Flowchart Language. *Partial Evaluation – Practice and Theory DIKU 1998 International Summer School, Springer Verlag*, 1706:20 – 82, Jun 1998.

[11] V. Itkin. On Partial and Mixed Program Execution. In *Program Optimization and Transformation*, pages 17–30. CCN, 1983. (In Russian).

[12] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[13] Kamen Yotov and Xiaoming Li and Gan Ren et.al. A Comparison of Empirical and Model-driven Optimization. In *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2003.

[14] R. Marlet, S. Thibault, and C. Consel. Efficient Implementations of Software Architectures via Partial Evaluation. *Automated Software Engineering: An International Journal*, 6(4):411–440, October 1999.

[15] Object Management Group. *Real-time CORBA Specification*, OMG Document formal/02-08-02 edition, Aug. 2002.

[16] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, Apr. 2000.

[17] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.

[18] C. Pu, T. Autery, A. Black, C. Consel, C. Cowan, J. W. Jon Inouye, Lakshmi Kethana, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium of Operating System Principles*, Copper Mountain Resort, Colorado, Dec. 1995.

[19] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[20] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale. Applying Optimization Patterns to the Design of Real-time ORBs. In *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, pages 145–159, San Diego, CA, May 1999. USENIX.

[21] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale. Using Principle Patterns to Optimize Real-time ORBs. *IEEE Concurrency Magazine*, 8(1), 2000.

[22] I. Pyarali and D. C. Schmidt. An Overview of the CORBA Portable Object Adapter. *ACM StandardView*, 6(1), Mar. 1998.

[23] I. Pyarali, D. C. Schmidt, and R. Cytron. Techniques for Enhancing Real-time CORBA Quality of Service. *IEEE Proceedings Special Issue on Real-time Systems*, 91(7), July 2003.

[24] F. v. d. L. Rob van Ommering, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 3(33):78–85, Mar. 2000.

[25] R. J. Rodger. Jostraca: a template engine for generative programming. In *ECOOP 2002 Workshop on Generative Programming*. Springer Verlag, 2002.

[26] W. Roll. Towards Model-Based and CCM-Based Applications for Real-Time Systems. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Hakodate, Hokkaido, Japan, May 2003. IEEE/IFIP.

[27] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts, 2002.

[28] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.

[29] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[30] D. C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[31] D. C. Sharp and W. C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[32] C. Zhang and H. Jacobsen. Re-factoring Middleware with Aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, Nov 2003.