COMPUTER SCIENCE

OPTIMIZATION TECHNIQUES FOR ENHANCING MIDDLEWARE QUALITY OF SERVICE FOR SOFTWARE PRODUCT-LINE ARCHITECTURES

ARVIND S. KRISHNA

Dissertation under the direction of Professor Douglas C. Schmidt

Product-line architectures (PLA)s are an emerging paradigm for developing software families by customizing reusable artifacts, rather than hand-crafting the software from scratch. In this paradigm, each product variant is assembled, configured, and deployed based on specifications of the required features and service-level agreements. To reduce the effort of developing software PLAs and product variants, it is common to leverage general-purpose – ideally standard – middleware platforms. These middleware platforms provide reusable services and mechanisms (such as connection management, data transfer protocols, concurrency control, demultiplexing, marshaling/demarshaling, and error-handling) that support a broad range of application requirements (such as efficiency, predictability, and minimizing end-to-end latency). A key challenge faced by developers of software PLAs is how to optimize standards-based – and thus largely application-independent – middleware to support the application-specific quality of service (QoS) needs of different product variants created atop a PLA.

This dissertation provides four contributions to research on optimizing middleware for PLAs. First, it describes the evolution of optimization techniques for enhancing application-independent middleware to support the application-specific QoS needs of PLAs. Second, it presents a taxonomy that categorizes the evolution of this research

in terms of (1) applicability, i.e., are the optimizations applicable across variants or specific to a variant, and (2) binding time, i.e., when are the optimizations applied during the middleware development lifecycle. Third, this taxonomy is applied to identify key challenges that have not been resolved by current research on PLAs, including reducing the complexity of subsetting, configuring, and specializing middleware for PLAs to satisfy the QoS requirements of product variants. Finally, the dissertation describes the OPTEML solution approach that synergistically addresses key unresolved research challenges via optimization strategies that encompass pattern-oriented, model-driven development, and specialization techniques to enhance the QoS and flexibility of middleware for PLAs. These optimizations have been prototyped, integrated, and validated in the context of several representative applications using middleware developed with Real-time Java and C++.

Approved _____ Date _____

OPTIMIZATION TECHNIQUES FOR ENHANCING MIDDLEWARE QUALITY OF SERVICE FOR SOFTWARE PRODUCT-LINE ARCHITECTURES

By

Arvind S. Krishna

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December, 2005

Nashville, Tennessee

Approved:

Professor Douglas C. Schmidt

Professor Aniruddha Gokhale

Professor Janos Sztipanovits

Professor Gabor Karsai

Professor Adam Porter

To my wife, Sriranjani, for always being there for me
and
To my parents for their support and prayers

ACKNOWLEDGMENTS

This quest for knowledge would not have been possible without the encouragement, support and guidance from researchers and practitioners in academia and industry. I am ever grateful to the following individuals for their help with my graduate education in the University of California, Irvine and Vanderbilt University.

First, I would like to thank my advisors and mentors. Dr. Douglas C. Schmidt has been my advisor and guide during my graduate education. Learning from and working with Dr. Schmidt has been a rewarding experience both professionally and personally. I am very thankful to him for guiding me through the various phases of graduate education. Dr. Aniruddha (Andy) Gokhale was my co-advisor and helped me in my research on Model-driven techniques. Andy's help has been important in concretizing the research ideas and translating them to conference/journal publications. The Quality Assurance research in this dissertation was guided by Dr. Adam Porter and his group at University of Maryland. His great sense of humor made our research discussions and teleconferences memorable. Dr. John Hatcliff at Kansas State University helped me with the specialization work presented in this dissertation. Dr. Raymond Klefstad was my co-advisor in the University of California, Irvine and provided me the opportunity to work with him on the ZEN project. I am also grateful to my dissertation defense committee Dr. Aniruddha Gokhale, Dr. Gabor Karsai, Dr. Adam Porter, Dr. Douglas C. Schmdit and Dr. Janos Sztipanovits for their time and effort spend-spent on reviewing and suggesting improvements to this dissertation. Thanks are also due to Dr. Michael Stal from Siemens Corporate Technology who read an earlier version of this dissertation and suggested improvements to the related work and dissertation format.

I am thankful to the following agencies provided financial support for research conducted in this dissertation. The research conducted on ZEN was funded by DARPA

program. Research on Model-driven development was funded by AFRL #F33615-03-C-41. Research on Skoll and Quality Assurance processes was funded by NSF and ONR #N00014-05-1-0421S. The specialization research was funded by a grant from Qualcomm. Other corporate sponsors include Lockheed Martin Eagen, Raytheon (Internal R&D) Rhode Island.

Over the past five years, I had the good fortune to have interacted and learned from several members of the DOC group. I would like to thank Dr. Carlos O' Ryan and Ossama Othman for helping me understand CORBA concepts and the intricacies involved with building middleware when I first joined the DOC group in Irvine. Their comments and guidance were crucial to the design of the ZEN POA. Discussion sessions with Dr. Angelo Corsaro helped me integrate Real-time Java features in ZEN. After moving to Nashville, Balachandran (Bala) Natarajan has been a constant source of guidance and support. The FOCUS tool and specialization strategies discussed in this paper would not have been possible without his suggestions. Bala also helped me understand the design of the TAO ORB. Dr. Nanbor Wang and Dr. Diego Sevilla Ruiz helped me understand the workings of different CCM implementations which was integral to the design of CCMPerf benchmarking suite. The work on middleware specializations techniques was motivated by Gary Daugherty at Rockwell Collins during the DARPA PCES program. Dr. Venkatesh Prasad Ranganath helped me to come up and implement the specialization techniques to TAO. Dr. Atif Memon and Dr. Cemal Yilmaz helped me in integrating BGML with the Skoll Quality Assurance tool.

An integral part of any Ph.D is the lasting relationship with one's cohorts. The discussions, debates, joys, frustrations and happy hours that I have shared with the following individuals Jaiganesh Balasubramanian, Krishnakumar Balasubramanian, Gan Deng, Amogh Kavimandan, Jeff Parsons and Nishanth Sankaran at Vanderbilt

University and Mayur Deshpande, Priyanka Gontla, Malli, Mark Panahi, and Krishna Raman at University of California Irvine will be fresh in my memory for a long time.

I will always be indebted to my parents for their love, support and encouragement all my life. Last but not the least, to my wife Sriranjani, without her I could not have come this far.

Arvind S. Krishna

Vanderbilt University

Nov 2005

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Software development processes are increasingly becoming demanding. For example, there is a growing need for software development organizations to innovate rapidly, provide capabilities that meet their customer needs, and sustain their competitive advantage. Adding to these demands are increasing time-to-market pressures and limited software resources, which often force organizations to innovate by leveraging existing artifacts and resources rather than hand-crafting software products from scratch. *Product-line architectures* (PLAs) [12] and *middleware* [79] are promising technologies for addressing these demands.

In contrast to conventional software processes that produce separate point solutions, in a PLA-based process, a family of product variants [89] is developed to share a common set of capabilities, patterns, and architectural styles. For example, Figure I.1 illustrates a portion of the Boeing Bold Stroke avionics mission computing PLA [91], which is designed to support a family of Boeing aircraft, including many variants of F/A-18, F-15, A/V-8B, and UCAV. Bold Stroke is a component-based, publish/subscribe platform built atop Real-time CORBA [61] and heavily influenced by the Lightweight CORBA Component Model (CCM) [60, 75].

PLAs in general – and Bold Stroke in particular – can be characterized using the *Scope, Commonality, and Variabilities* (SCV) analysis [13]. SCV is a domain engineering process that identifies common and variable properties of an application domain. Domain/systems engineers and software architectures use this information in the SCV process to guide decisions about where and how to address possible variability and where the common development strategies can be used.

Applying the SCV process to Bold Stroke yields:

- **S**, *e.g.*, the scope is Boeing's component architecture and associated set of components that address the domain of avionics mission computing, which includes services such as heads-up display, navigation, auto-pilot, targeting, and sensor management.
- **C**, *e.g.*, the commonalities are the set of common components and connections between, such as connection management, data transfer, concurrency, synchronization, demultiplexing, error-handling, etc. that occur in all product variants.
- **V**, *e.g.*, the variabilities include how various subsets of components are connected together to support the requirements of different customers (such as F/A-18E vs. F-15K), their different implementations (such as which algorithms are chosen for each product variant), and components that are specific to a variant (such as restrictions due to foreign military sales).

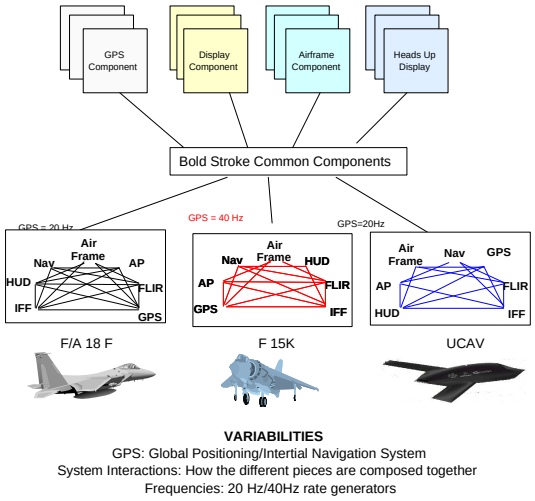

Figure I.1: SCV Analysis for Boeing Bold Stroke PLA

After a PLA has been developed and has matured, the ensuing development of product variants ideally proceeds in top down manner. Figure I.2 illustrates the process of developing a product variant, which starts with a clear statement of the

required capabilities and QoS. Higher level models and analysis tools [30, 34] compose, analyze, and validate the product-line to ensure semantic compatibility. The next step involves the composition of a variant from existing components from the repository. This phase also involves mapping of the requirements on to PLA artifacts, such as communication protocols, service-level agreements, and configuration/deployment policies/mechanisms. Finally, the system is deployed on a platform such as CORBA [62], J2EE [94] or .NET [56].

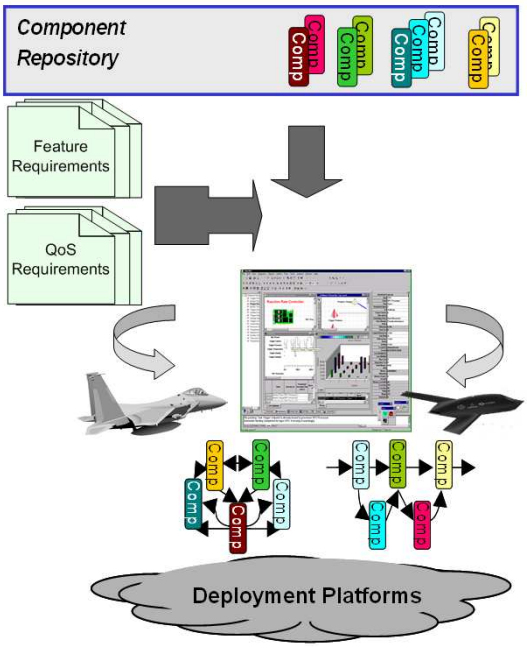

Figure I.2: PLA Development Process

Although PLAs can be developed and applied to many domains, an increasingly important domain for applying PLAs is *distributed, real-time and embedded (DRE) systems* [19, 89, 91]. Examples of DRE systems include applications with hard real-time requirements, such as avionics mission computing [87], as well as those with softer real-time requirements, such as telecommunication call processing and streaming video [86]. These types of systems are characterized by their multiple, simultaneous constraints across different QoS dimensions (such as memory footprint, weight,

and performance), which often makes them harder to develop, maintain, and evolve than mainstream desktop and enterprise software. These challenges have hitherto forced developers of DRE systems to repeatedly reinvent custom solutions that are tightly coupled to specific hardware and platforms, which is tedious, error-prone, and costly over product lifecycles.

A key enabling technology for developing and customizing PLAs is *middleware*, which is systems software that resides between the application and the underlying operating system that (1) functionally bridges the gap between application program and lower-level hardware and (2) simplifies the integration of components developed by multiple technology suppliers [79]. During the past decade, quality of service (QoS)-enabled middleware has emerged to help developers of DRE systems (1) factor out reusable concerns (such as component lifecycle management, authentication/authorization, and remoting) to enhance reuse and (2) shield from low-level tedious, error-prone, and non-portable platform details, such as socket and threading programming.

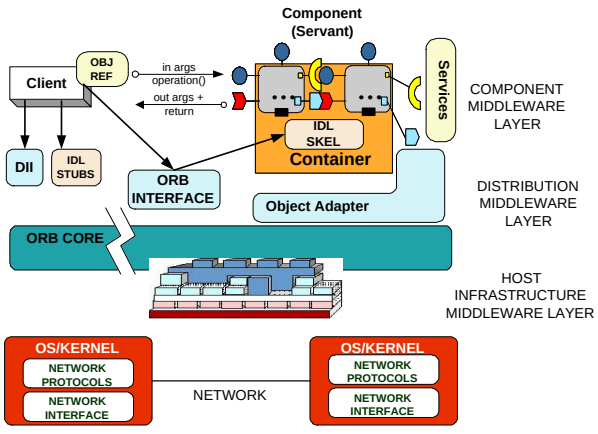

Figure I.3: Layered Middleware Architecture

Figure I illustrates a widely applied middleware architecture [27] that underlies

PLAs used for DRE systems [27, 49, 89–91]. This figure illustrates two key characteristics of middleware:

- **Design for generality**, where each layer is designed to host different applications. For example, the Java Virtual Machine (JVM) [48] is middleware that provides concurrency, synchronization, serialization, and messaging portably via common set of API across a wide range of platforms.
- **Layered architecture**, where different middleware layers are stacked to address end-to-end QoS needs. For example, CORBA is a standard distribution middleware layer that provides network programming capabilities (such as connection management, data transfer protocols, concurrency control, demultiplexing, marshaling/demarshaling, and error-handling) and location transparency to applications.

Standards-based QoS-enabled middleware technologies, such as Real-time CORBA [61] and Real-time Java [6], support the provisioning of key QoS properties, such as (pre)allocating CPU resources, reserving network bandwidth/connections, and monitoring/enforcing the proper use of DRE system resources at runtime to meet end-to-end QoS requirements, such as throughput, latency, and jitter. QoS-enabled component middleware technologies, such as Lightweight CCM [60] and Prism [89], simplify QoS provisioning via metadata and tools that help to (1) automate DRE system development lifecycle phases, such as packaging, assembly, configuration, and deployment, and (2) improve component reusability and performance by preventing premature commitment to specific QoS provisioning decisions, such as allocating components to thread pools and selecting the underlying transport protocols. As a result, software for DRE systems is increasingly being assembled from reusable modular components in PLAs using standards-based middleware platforms, rather than hand-crafted manually from scratch.

Research Challenges

Although middleware is a crucial technology for PLAs, key challenges must be overcome before it can be applied seamlessly to support the QoS needs of DRE systems developed using PLAs. In particular, Figure I.4 illustrates the current tension between (1) application-specific product variants, which require highly-optimized and customized PLA middleware implementations and (2) general-purpose, standards-based, reusable middleware, which is designed to satisfy a broad range of application requirements.

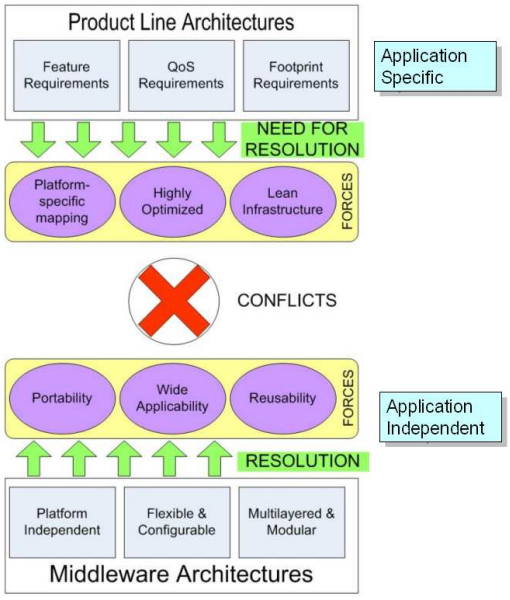

Figure I.4: Application-specific vs. Application-independent Dimensions of PLAs and Middleware

Resolving this tension is essential to ensure that middleware can support the QoS requirements of DRE systems developed using PLAs. Unfortunately, even today's leading standards-based QoS-enabled middleware technologies, such as Real-time CORBA [61] and Real-time Java [6] are not yet capable of supporting PLAs for DRE systems due to the following limitations:

1. Monolithic middleware implementations that include more capabilities than are needed for particular product variants. Standards-based middleware for PLAs is often implemented in a monolithic "one-size-fits-all" manner, *i.e.*, it includes code supporting many mechanisms (such as connection and data transfer protocols, concurrency and synchronization management, request and operation demultiplexing, marshaling/demarshaling, and error-handling), even when this code is not used/needed. A key research challenge is therefore making middleware extensible to enable the selection of necessary middleware mechanisms.

2. Overly general middleware implementations that incur excessive time and space overhead for particular product variant use cases. Standards-based middleware is designed for generality, *i.e.*, its capabilities support a range of applications, *e.g.*, CORBA middleware supports many different types of applications running over many different types of transports. However, standards-based middleware often incurs excessive generality imposed by the standard. For example, (de)marshaling for standard CORBA incur byte order test overhead, even if the machines on which they are hosted conform to the same hardware instruction set. A key research challenge is therefore to use ahead-of-time properties for each product-line variant to specialize middleware.

3. Ad hoc techniques for validating and understanding how middleware configurations influence end-to-end QoS. Middleware for PLAs often provides a range of options that can be parameterized into various configurations. Many of these settings (such as concurrency strategies, buffer sizes and locking) directly affect end-to-end QoS. It can be hard, however, to tune and validate the QoS properties of such configurable middleware. Product variants often use *ad hoc* approaches to identify the right set of middleware configurations that satisfy the system end-to-end latency and QoS requirements. Moreover the process they use is not repeatable (reusable across different variants) and suffers from accidental complexities stemming from the need

to write low level source code (XML configuration files, interface declaration and QoS and benchmarking code) for capturing impact of middleware configurations on QoS. A key research challenge is therefore devising a systematic approach for evaluating, validating, and capturing impact of middleware configurations on end-to-end QoS.

In summary, key challenges that remain to be addressed center on developing and validating technologies and tools for (1) capturing application-specific requirements of particular product variants and (2) using these requirements to drive the optimization of PLA middleware implementations to eliminate the time/space penalties associated with using general-purpose, standards-based, and reusable middleware for DRE systems. Resolving these challenges is essential to support the new generation of standards-based middleware that will be easy-to-use, extensible, and flexible, as well as providing the appropriate QoS to meet the needs of PLAs for DRE systems.

Research Approach

To address the challenges described in Section I, this dissertation has developed Optimization Techniques for Enhancing Middleware QoS for PLAs (OPTEML). The different dimensions of this approach are shown in Figure I.5 and described below.

1. Componentize PLA middleware at a fine level of granularity to include only capabilities required for each product variant. This approach factors out different middleware mechanisms into modular pluggable components that are not loaded until they are used. Each factored service itself can in turn be considered as a monolithic element and factored out into modular components at a finer level of granularity. In particular, the contribution of this portion of the dissertation is the fine-grain componentization of the CORBA *Portable Object Adapter (POA)* [71] using policy-driven approaches. Chapter III describes this approach for fine-grained componentization of PLA middleware in detail.

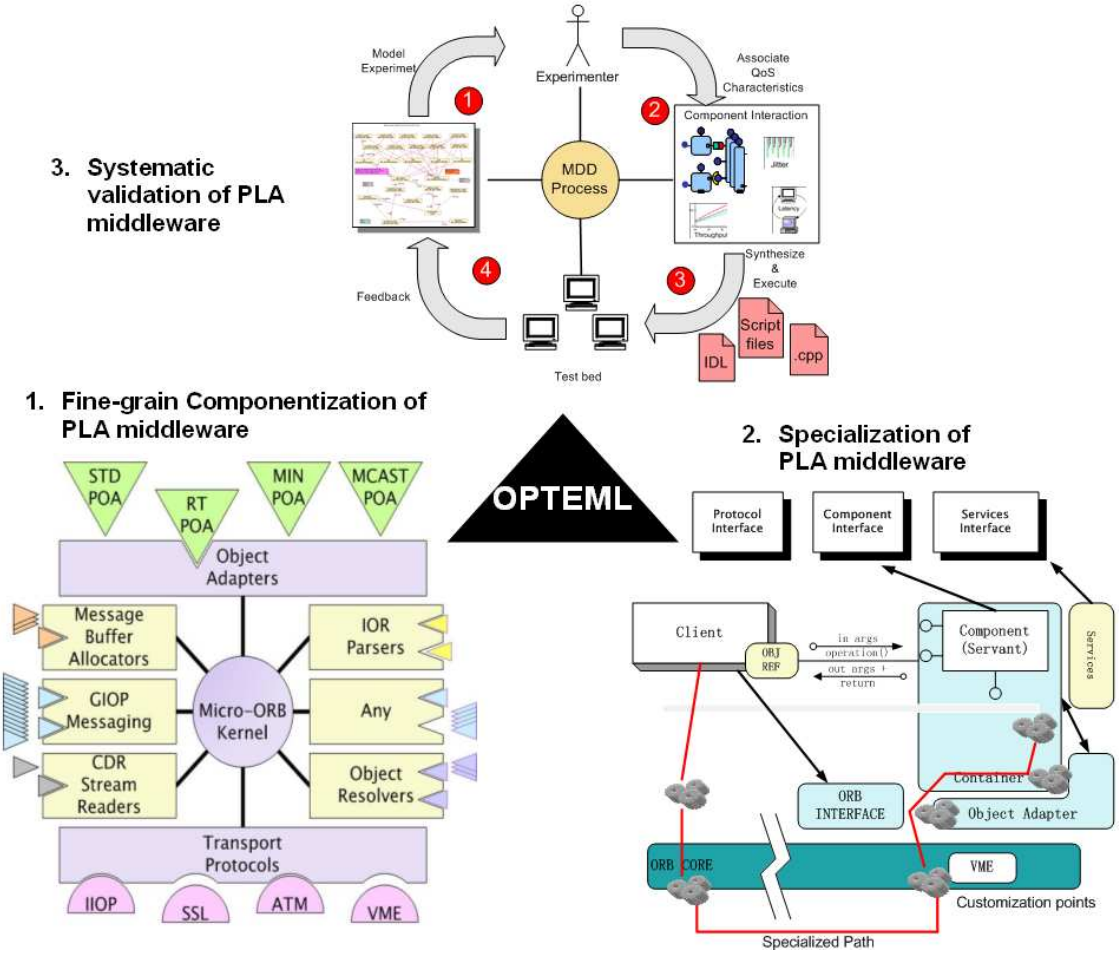

Figure I.5: Dimensions of OPTEML Research

2. Specialize PLA middleware to eliminate unnecessary time and space overhead. This approach focuses 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 [39]. Chapter IV describes this approach for specialization of PLA middleware in detail.

3. Systematically validate configurations of PLA middleware that satisfy the QoS requirements of product variants. This approach uses model-driven

development (MDD) techniques to capture the QoS requirements of product-line variants in higher level models and synthesize validation code, include (1) the XML configuration settings that are to be evaluated, (2) the XML deployment data that will be used to deploy the component on to target platform, and (3) the QoS evaluation and benchmarking code that will measure the QoS and identify the right configurations that maximize QoS. This MDD approach is combined with advanced statistical techniques to evaluate empirically how specialized and general-purpose optimizations of middleware affect end-to-end QoS for different PLAs and product variants. Chapter V describes the proposed approach for validating PLA middleware configurations in detail.

Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter II presents a taxonomy of existing research efforts that are related to OPTEML and uses the taxonomy to show how the key challenges discussed in Section I have not been addressed adequately in existing research on PLAs; Chapter III illustrates how fine-grain middleware componentization techniques can be applied to customize monolithic middleware implementations thereby minimizing middleware footprint and facilitating ease of adaptation; Chapter IV describes how context-specification specialization techniques can be applied 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; Chapter V shows how model-driven distributed continuous QA tools are synergistically combined to evaluate how general-purpose and specialized middleware optimizations affect PLA QoS; and Chapter VI presents concluding remarks, discusses research contributions, impact and outlines future work.

CHAPTER II

RESEARCH EVOLUTION

This section systematically explores and documents research addressing different issues relating to OPTEML. To structure the discussion, a taxonomy, *i.e.*, classification, is presented that categorizes related research across the following two dimensions

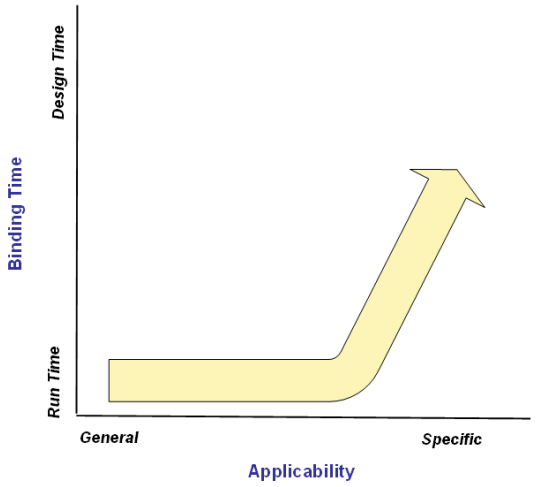

Figure II.1: Research Taxonomy

shown in Figure II.1:

- **Applicability**, *i.e.*, research contributions that are broadly applicable (general) across different product-lines architectures versus techniques that are applicable to only a given variant.
- **Binding time**, *i.e.*, research that deals with optimizations that are applied at run-time versus optimizations that are built into the middleware at design-time or at deployment time.

The use of the taxonomy allows research to be categorized into an evolving *continuum* of optimizations (progressing from general-purpose optimizations to more application specific design-time optimizations) that are described below:

- **General-purpose optimizations**, that classify research on algorithmic and data structural optimizations that have been applied at different layers of middleware to improve performance.
- **Configuration-driven optimizations**, that classify research on analysis techniques that evaluate and quantify impact of different software configuration settings on product-line level QoS.
- **Specialization optimizations**, that classify research on program optimization and specialization techniques that modify software implementation based on ahead of time (AOT) binding software configuration parameters.

The remainder of this chapter is organized as follows: For each class of optimization, a succinct description of related research is presented. Each section then describes the research areas that require resolution.

General-purpose Optimizations

Research on customizing middleware for different PLAs originated with research in the early 1990's on how to optimize middleware to improve performance. As illustrated in Figure II.2, this research greatly focused on examining different data structures and algorithmic optimizations that can be applied at different layers, such as operating systems, network protocols and middleware layers to improve application QoS. These optimizations are not applied ad hoc, but ultimately lie along the critical request/response path of QoS enabled middleware implementations. Such optimizations addressed application concerns such as end-to-end predictability, scalability and latency/throughput. This section categorizes this body of knowledge as

general-purpose optimizations as these techniques are very generic, *i.e.*, these optimizations can be leveraged universally across different product-line variants.

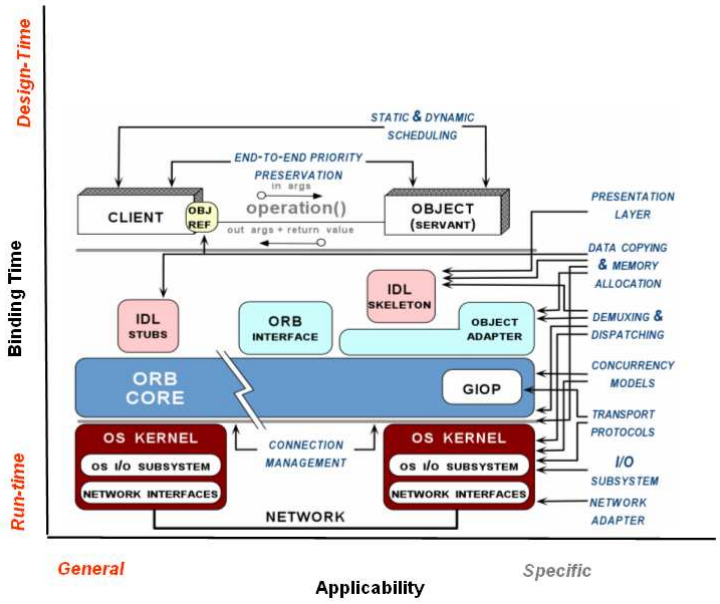

Figure II.2: General-purpose Optimizations

In addition, these optimizations are applied at run-time and fall into our taxonomy as **general-purpose run-time optimization techniques**.

Dimensions of General-purpose Optimizations

Research on general-purpose optimizations can be categorized based on two principal activities that are essential for improving performance, and scalability of applications. These include, (1) efficient and optimized layer-to-layer demultiplexing techniques that find the target servicing each request and (2) optimal concurrency strategies, that determine the number of such request that can be processed simultaneously and efficiently. Research on these two dimensions are explained below:

Request Demultiplexing approaches. Research on improving demultiplexing performance has focused on eliminating layered demultiplexing approaches in both the

protocol stack and within middleware. For instance, [18, 23, 97] study demultiplexing issues in communication systems and show how layered demultiplexing is not suitable for applications requiring real-time quality of service guarantees. Packet filters are a mechanism for efficiently demultiplexing incoming packets to application endpoints [57]. A number of schemes to implement fast and efficient packet filters are available. These include the BSD Packet Filter (BPF) [54], the Mach Packet Filter (MPF) [104], PathFinder [3], demultiplexing based on automatic parsing [38], and the Dynamic Packet Filter (DPF) [22].

In the CORBA middleware, research efforts have focused on ensuring $O(1)$ demultiplexing time bound for different layers within CORBA middleware. Perfect hashing [83] is a technique that generates collision free hash functions when the keys to be hashed are known *a priori*. In many hard real-time systems (such as avionic control systems [32]), the objects and operations can be configured statically. Research effort in [29] has used perfect hashing to generate hash functions for operation names defined in IDL. In other efforts [69], de-layered active demultiplexing strategy is used to flatten hierarchy and locate the target object in one table lookup.

Concurrency approaches. Concurrency strategies describe how multiple tasks will be executed simultaneously. For web servers or CORBA servers, a task is a set of server request handling steps. Several concurrency strategies, such as iterative, single-threaded, thread-per connection and thread-pool strategies have been applied in web and CORBA servers.

Research in [53] measured the impact of synchronization on Thread-per-Request implementations of TCP and UDP transport protocols built within a multi-processor version of the *x*-kernel; [58] examined performance issues in parallelizing TCP-based and UDP-based protocol stacks using a Thread-per-Request strategy in a different multi-processor version of the *x*-kernel; and [78] measured the performance of the TCP/IP protocol stack using a thread-per-connection strategy in a multi-processor

version of System V STREAMS. The ADAPTIVE [7] framework examines research on parallelizing transport architectures.

Table II.1: Dimensions of General-purpose Optimizations

General-purpose Optimization Alternatives	
Lookup-based	*Alternatives*
fixed	perfect-hashing
dynamic	linear-search, dynamic hashing, active demultiplexing
Concurrency	*Alternatives*
request-processing models	asynchrony (client/server side), synchrony
threading models	thread-per request, thread-per connection, thread-pool
Domains	*Implementations*
protocol	ADAPTIVE
application/web servers	JAWS, POSA patterns

The JAWS [35] framework provides different concurrency strategies for building high performance web-servers. Research work on concurrency ties closely with issues of synchronous versus asynchronous request processing. Research in [20] has looked into implementation of *continuations* in the MACH kernel, which decouples the request demultiplexing from the processing. In ORBs, Asynchronous Method Handling (AMH) [17] provide similar mechanisms like continuations. On the client side, the Asynchronous Method Invocation (AMI) [10] ameliorates clients blocking overheads when waiting for reply for a long running request.

Summary. The research efforts discussed earlier and related efforts documented in [80–82] to better implement high performance architectures have distilled into systematic body of knowledge in the form of design patterns [26]. The Pattern Oriented Software Architecture (POSA) 2 [88] book describes a pattern language for building concurrent high performance servers by discussing patterns for service access and configuration, event handling, synchronization and concurrency. Table II categorizes different general-purpose optimizations across three dimensions. The first dimension categorizes research on request demultiplexing strategies into fixed (constant time and space) versus variable (variable time/space) strategies. The second dimension

categorizes research on concurrency, *i.e.*, request processing models synchronous and asynchronous processing and threading models. Finally, the table illustrates the different domains on which these optimizations have been applied.

What Remains to be Done

Traditional general-purpose optimizations have looked at performance related issues of middleware. These have attained maturity and also have demonstrated middleware as a mature solution for DRE product-line systems. However, an orthogonal issue to performance is the concern of componentizing the ORB services at a fine-grain level to allow product-line variants to select the set of middleware features. This concern is accentuated by DRE product-line architecture size requirements. As *embedded systems*, DRE systems have weight, cost, and power constraints that limit their computing and memory resources. For example, embedded systems often cannot use conventional virtual memory, since software must fit on low-capacity storage media, such as EEPROM or NVRAM.

The evolution of middleware has resulted in architectures that are inherently *monolithic*, in which all middleware features reside in a single executable. Static mechanisms/tools, such as conditional compilation and smart static linkers, allow a variety of different configuration options. This approach is harder to code and maintain due to accidental complexities involved with conditional compilation [44]. Further achieving a small footprint is possible only if the architecture is initially designed to achieve it. It is much harder to reduce footprint in later stages of design. Chapter III describes in detail how this limitation is resolved using the OPTEML approach.

In addition to the fine-grain componentization concerns, general-purpose optimizations are exposed as configurable and tunable knobs. For mature middleware implementations, such as ACE+TAO [36] open-source middleware, this has resulted

in an exponential increase in the number of configuration settings[1]. This trend requires product-line variants to understand how the interplay between middleware configurations affect and influence end-to-end QoS. The next section describes how configuration-driven optimization techniques are addressing some of these issues.

Configuration-driven Optimization Techniques

In middleware implementations, general-purpose optimization techniques are exposed as middleware configuration settings that can be enabled/disabled at build/run time. These options therefore require product-line architects to understand the trade-off, in terms of application QoS, accrued by enabling/disabling these configuration settings. This dependency is akin to the optimization settings that can be used with an optimizing compiler. For example gcc [25], which provides a compiler suite, allows setting and un-setting different configuration knobs that optimize for speed (-O3, -02 options), size (-Os) or different processor architectures (all options that have -m perpended).[2]

Similar to how one needs to understand the consequences of enabling different compiler options, Figure II.3 illustrates the need for product-line variants to understand the consequences of middleware settings. This problem is exacerbated by the fact that (1) not all combinations of middleware options form a semantically compatible set, and (2) match the application QoS requirements onto middleware configuration settings to maximize QoS. **Configuration-driven optimizations** are techniques to tune middleware configuration knobs to maximize application QoS. These techniques are bound either at run-time via online techniques or are evaluated by Quality Assurance (QA) engineers at test/evaluation time.

[1]Examples of highly configurable middleware in other domains include (1) SQL Server 7.0, which has ∼50 configuration options, (2) Oracle 9, which has over 200 initialization parameters, and (3) Apache HTTP Server Version 1.3, which has ∼85 core configuration options.

[2]A comprehensive list of compiler options for gcc are available from: `http://gcc.gnu.org/onlinedocs/`.

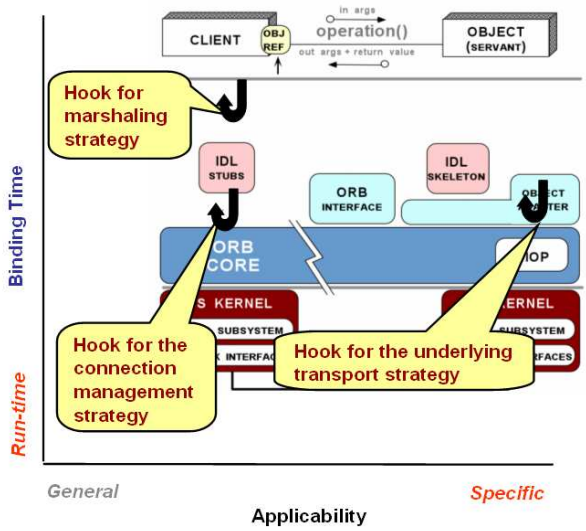

Figure II.3: Configuration-driven Optimizations

These optimization techniques are applied on a per-application basis but themselves are generalizable across different product-line variants.

Dimensions of Configuration-driven Optimizations

This section builds a taxonomy, *i.e.*, categorizes configuration-driven optimization approaches into feedback-driven techniques (online, offline and hybrid techniques), techniques for QoS evaluation (generative programming approaches and performance patterns) and techniques for functional correctness of software configurations (testing approaches and test coverage).

Feedback-driven techniques. Feedback driven approaches can be categorized into the following three broad analysis techniques:

Offline analysis, which has been applied to program analysis to improve compiler-generated code. For example, the ATLAS [40] numerical algebra library uses an empirical optimization engine to decide 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 provide the best

performance. Mathematical models are also used to estimate optimization parameters based on the underlying architecture, though empirical data is not fed into the models to refine it.

Online analysis, where feedback control is used to dynamically adapt QoS measures. An example of online analysis is the ControlWare middleware [106], which uses feedback control theory by analyzing the architecture and modeling it as a feedback control loop. Actuators and sensors then monitor the system and affect server resource allocation. Real-time scheduling based on feedback loops has also been applied to Real-time CORBA middleware [50] to automatically adjust the rate of remote operation invocation transparent to an application.

Hybrid analysis, combines aspects of offline and online analysis. For example, the continuous compilation strategy [11] constantly monitors and improves application code using code optimization techniques. These optimizations are applied in four phases including (1) *static analysis*, in which information from training runs is used to estimate and predict optimization plans, (2) *dynamic optimization*, in which monitors apply code transformations at run-time to adapt program behavior, (3) *offline adaptation*, in which optimization plans are actually improved using actual execution, and (4) *recompilation*, where the optimization plans are regenerated.

Functional correctness based techniques. The following are different approaches for evaluating the correctness of software across different configurations:

The MODEST [76] tool provides a generative approach for producing (1) test cases, *i.e.*, test-code that is used to test the system and (2) test-harness, *i.e.*, the scaffolding code required for test setup and tear down. In MODEST, test cases are generated in parallel with the actual system. The motivation being to provide the users with not only the system but also the test-code to reduce maintenance costs.

SoftArch/MTE [31] provides a framework for system architects to define higher level abstractions of their system by specifying characteristics such as middleware,

database technology, and client requests. SoftArch/MTE then generates an implementation of the system along with the performance tests that measure these system characteristics. These results are then displayed (*i.e.*, annotated in the high level diagrams) using tools such as Microsoft Excel, thereby allowing architects to refine the design for system deployment.

Skoll [55] is a distributed continuous quality assurance (DCQA) tool for developing and validating novel software QA processes and tools. Skoll leverages the extensive computing resources of worldwide user communities in a distributed, continuous manner to significantly and rapidly improve software quality. In particular, Skoll provides an integrated set of technologies and tools that run coordinated QA activities around-the-world, around-the-clock on a virtual computing grid provided by user machines during off-peak hours.

Techniques for configuration tuning. The following are different approaches for quantifying impact of software configuration on QoS:

Performance Patterns [59] and Performance Pattern Languages (PPL) provide an automatable, script-based framework within which extensive ORB endsystem performance benchmarks can be described efficiently and executed automatically. These patterns are embodied in the NetSpec tool developed at the Kansas University. The patterns themselves are written using PPL. Examples of such patterns include Cubit Tests (measuring (de) marshaling overhead), Client-Server benchmarks (simple client server two node approach for benchmarking) and Proxy benchmarks (introducing a proxy that acts as an intermediary between the client and server).

Performance tuning approaches in [92] are examining the importance of webservice parameters for different workloads and request types using statistical techniques. Their approach analyzes how end-to-end performance varies for different commonly used web service requests, *e.g.*, buy request, product query and search requests. For

each kind of request, they use different workloads published in the web-service benchmarks, *e.g.*, varying the user think time and number of users. An initial configuration space is chosen and different possibilities of each configuration setting are exercised to measure their influence on end-to-end quality of service. This data is used as input in the ANOVA [47] analysis to determine the statistical significance of different web-server options.

Summary. Table II categorizes configuration driven approaches along two dimensions. The first dimension describes

Table II.2: Dimensions of Configuration-driven Specialization Mechanisms

Configuration-Driven Specialization Alternatives	
Time-based	*Alternatives*
online	Controlware middleware
offline	ATLAS, MODEST, Soft/ARCH
hybrid	continuous compilation, Skoll
guidance	performance patterns, ANOVA
Type of Exploration	*Alternatives*
manual	performance-patterns and regression tests
automated	Skoll, continuous compilation, ATLAS
hybrid	generative-approaches mapped onto automated frameworks
Cost-based	*Alternatives*
none	continuous compilation, ANOVA, performance-patterns, Skoll
amortized	ATLAS
non-trivial	controlware

the approaches based on time. An offline approach is run a priori while an online approach is turned on while the software is run. In a guidance approach, data collected either from online/offline approach is used to build body of knowledge, which in turn is used to guide configuration selection. The second dimension compares the technique used for configuration exploration. For example, a manual approach may exhaustively or minimally try to explore the different configuration knobs. Automated/hybrid approaches may use techniques to build a configuration model or generate the right configuration space required for evaluation.

The final dimension explores the cost of the configuration space exploration. An

offline technique does not incur any run-time cost, whereas the online technique incurs a non trivial overhead for monitoring performance.

What Remains to be Done

The configuration-driven optimization techniques present online, offline, empirical and statistical tools for mapping higher level application concerns on to software configurations. These techniques however suffer from:

- **Repeatability limitations**, which stymie each product-line variant to execute the same process [65] to identify the pertinent configuration settings,
- **Cost limitations**, which increase the accidental complexities involved in actually handcrafting the scaffolding code for different product-line variants to map QoS requirements onto middleware configurations, and
- **Validation limitations**, which prevent the validation of these approaches across different platforms, hardware, compiler and OS options.

Earlier efforts on benchmarking distribution middleware implementations [42, 43] identified how tedious and error-prone the process of evaluating these configurations really were. Chapter IV describes in detail how this limitation is resolved using the OPTEML approach.

Another significant limitation of these techniques is that middleware or software cannot be tailored or customized once the required configuration knobs are determined. For example, if the right concurrency strategy to be used within middleware is determined to be single-threaded, this approach cannot remove the no-implementation locking code from within the middleware. The next section discusses techniques that can be applied to resolve this limitation.

Specialization Optimizations

Jones et al. [39], define specialization as a technique that creates a specialized version of a general program, which is more optimized for speed and size than the original program. This technique draws from and has characteristics of language mechanisms such as program optimization techniques [37], compilers [1], program generation [98] and generative programming techniques [14]. Specializations tailor implementations using Ahead of Time (AOT) known system properties or *invariants*. These techniques are applied very early in the development process *i.e.*, at design time rather than at run-time. In addition, these techniques are very application

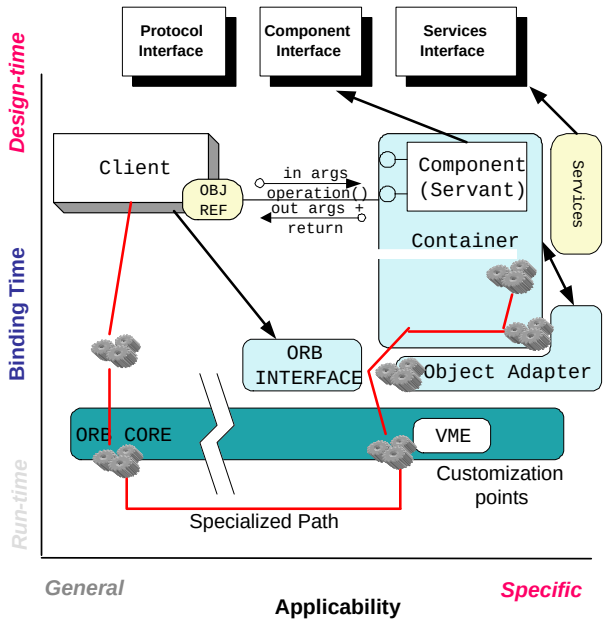

Figure II.4: Specialization Optimizations

specific. Figure II.4 illustrates how AOT properties can be used to specialize middleware.

Dimensions of Specialization Mechanisms

This section analyzes different dimensions of specializations by first building a taxonomy based on different mechanisms and then describing how these have been applied to different areas including operating systems, databases and neural networks.

Time-based mechanisms. Specialization can be carried out both *offline* and *online*. An offline technique occurs in two steps: In the first step, a *binding-time* analysis annotates the program code with static and dynamic information. This step is referred to as binding time analysis. In the second step, a code generator actually produces the optimized code. An online specialization uses the actual values directly rather than the two step process of annotation and program optimization. The online technique is more powerful than offline techniques as it deals with the actual value. However, offline specialization enables runtime adaptation as the information is already propagated within the code. Different tradeoffs are listed in [21].

Language mechanisms. Common specialization mechanisms at the language level include approaches that are two level in nature as described in [99]. Common examples include, macros (C and C++ language) for code expansion and templates in C++. In these examples, the code expansions are explicitly programmed. In the templates approach, each optimization, *i.e.*, a specialization can be explicitly programmed providing a fine grained control. This serves as its bane because optimizations have to be explicit. C++ templates are offline mechanism, the reason being, during the, first pass the compiler does not generate code but checks the syntax of the template code. The specializations are instantiated only when used. Macros are online specializations as they directly substitute code with no code annotation.

Specification-level mechanisms. Specification level specialization approaches include, code generators such as CORBA IDL compilers [100] and rpcgen [93] (IDL compiler for Sun RPC). These code generators generate glue-code for (de) marshaling

data, connection management and error handling. A common example is specialization based on object location where the IDL compiler generates special glue code of the objects that are within the same address space. Specification level specialization leverages language level specialization. For example, the generated code from a IDL to C++ compiler uses a specific marshaling routine implemented as generic templates.

Pattern matching mechanisms. Similar to specification level mechanisms, pattern matching approaches take regular expression as input and perform the specified action when there is a match. The matching serves as a mechanism of enabling specialization. For example, the code woven in via pattern matching can be partially specialized code rather than un-optimized code. The key difference here between pattern and language partial specialization approaches is that pattern matching can be global while language mechanisms are local. A common example in this category is AspectJ.

Application of specialization techniques. Specialization mechanisms have been applied to different domains including scientific applications, functional programming, operating systems and database systems. This section provides examples of specialization techniques that have been applied to optimize different algorithms. In computer graphics for example, ray tracing algorithms compute information on how light rays traverse a scene based on different origination. Specialization of these algorithms [2] for a given scene have yielded better performance rather than general purpose approaches. Similarly in databases [77], general purpose queries have been transformed into specific programs optimized for a given input. Similarly, training neural networks [46] for a given scenario has improved its performance.

In addition to the aforementioned domains, specialization techniques have also gained importance within the operating systems domain. The earliest of the efforts in Synthesis Kernel [67] pioneered the idea of generating custom system calls for specific situations. The motivation was to collapse layers and to eliminate unnecessary

Table II.3: Dimensions of Different Specialization Mechanisms

Specialization Alternatives	
Taxonomy	*Alternatives*
Time based	Online & Offline
Language based	Macros, templates, template meta-programming
Specification based	IDL compiler, rpcgen
Pattern based	AspectJ
Effort	manual (Language-based), automatic (pattern/specification based)
Domains	*Applications*
OS	Synthesis Kernel, HP_UX incremental specialization
Databases	dedicated read queries
Physics	Ray Tracing specialization
AI	Neural network specialization
Languages	*specialization tools*
Lambda Calculus	Lamdamix
Prolog	Logimix
Scheme	Similix
C	C-mix

procedure calls. Others have extended this approach to use incremental specialization techniques. For example in their work [68], Pu et al., have identified several invariants for a operating system `read` call for HP_UX platform. 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.

Summary. Table II categorizes specialization initiatives across three different dimensions. The first dimension shows the different specialization mechanisms. These approaches are then applicable to different domain as shown in the second dimension. Finally, the table shows several partial specialization tools that have been developed to specialize programs written in the corresponding languages. We do not discuss each of the tools in detail, more information about each tool is available in [39].

What Remains to be Done

Traditional specialization techniques have been used to optimize applications in function/logic programming. There does not exist any specialization tool for object oriented programming languages such as C++ or Java. Specialization variants, such as program specialization techniques are commonly used in optimizing compilers. Middleware displays several characteristics amenable to specialization such as (1) ability to run on different platforms, (2) multitude of configuration options and (3) design for flexibility and generality. Using a similar approach as an optimizing compiler, specialization may be used to produce leaner and meaner middleware implementations more tailored to the operating context

Middleware implementations traditionally are designed for generality *i.e.*, design facilitates use in different operating contexts. Middleware architectures are layered to support pluggable context-specific implementations. To improve performance and footprint, middleware implementations incorporate several horizontal (general purpose) optimizations such as predictable and scalable (1) request demultiplexing techniques, that ensure $O(1)$ look up time [41] and collocation optimization, which bypasses the network when client and server reside in the same address space. However, these optimizations are still generic, for example redundant checks for remoting are performed to accommodate for generality, *i.e.*, capability to communicate over the wire as well.

The configuration driven optimizations, help in choosing the right set of middleware configurations, however, do not eliminate the generality in middleware. Therefore, the research challenge is to explore the use of program specialization techniques to remove middleware generality. Chapter IV how this limitation is resolved using the OPTEML approach.

Summary

The research evolution and taxonomy presented in this chapter clearly show a trend from general-purpose run-time optimizations to highly applicable design time optimizations for enhancing middleware for PLA based application development. This chapter also described the *missing pieces* that have not been addressed by research approaches thus far. Addressing these principal challenges will herald the next generation of configurable and customizable middleware in the following manner.

- **Resolution of feature subsetting challenges,** will enable each product-line variant to select the middleware pieces that match the product line's feature requirements.
- **Resolution of specialization challenges,** will use the chosen configuration as drivers for removing middleware generality.
- **Resolution of configuration & validation challenges,** will enable the selection of right middleware configurations using a tool-driven repeatable process.

The remaining portion of this dissertation describes in detail how OPTEML addresses the aforementioned research challenges.

CHAPTER III

TECHNIQUES FOR FINE-GRAIN COMPONENTIZATION OF PLA MIDDLEWARE

A standards compliant CORBA ORB provides several services including support for multiple protocols, marshaling and demarshaling, multiple formats for exporting references and multiple object adapters that map client requests to implementation defined servants. Product-line variants then choose the set of middleware services that are required for satisfying their feature requirements. The taxonomy on general-purpose optimizations (described in Section II) identified limitations with existing middleware architectures including:

- **Lack of feature subsetting** – Implementing a full-service, flexible, specification-compliant ORB can yield a monolithic ORB implementation with a large memory footprint as shown in Figure III.1. Moreover, the footprint grows with each extension (adding support for a new protocol), and extensibility is hard.
- **Inadequate support for extensibility** – Designing middleware for PLAs not only require a full range of CORBA services but also need middleware to be adaptable, *i.e.*, meet the needs of wide variety of PLA applications developers. Current CORBA middleware designs are not designed with the aim of applicability in various domains.

Custom software development and evolution is labor-intensive and error-prone for complex DRE applications. Next generation middleware design should therefore be simultaneously extensible, provide feature subsetting, and be easy to use, thereby minimizing the the total system acquisition and maintenance costs. The remainder of this chapter describes novel techniques for fine-grain componentization of middleware for PLA. These techniques (1) enable different levels of middleware componentization,

e.g., *coarse-grain* (decomposing a monolithic component into sub components) and *fine-grain* (further decomposition of the already factored out pieces),

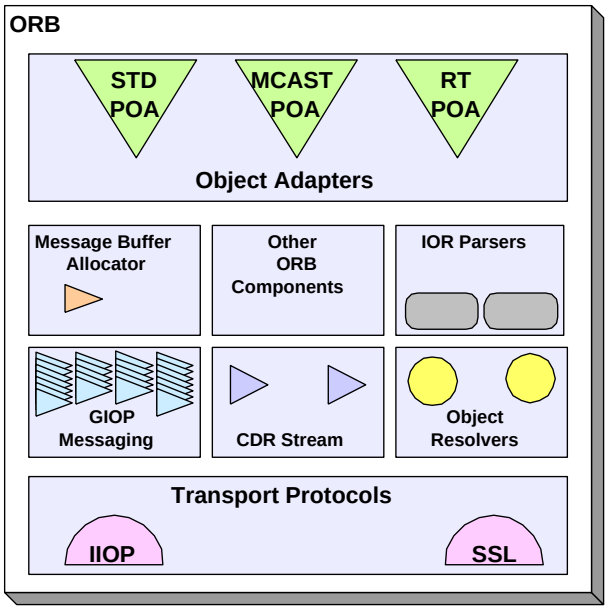

Figure III.1: Monolithic ORB Architecture

(2) are transparent to the application (no changes to the PLA application code), (3) significantly reduces middleware footprint and (4) allow easy addition of new features without sacrificing performance.

Micro ORB Designs

To enhance the customizability and flexibility of middleware implementations, an ORB should allow an application to select the minimal set of components required. To address the limitations with monolithic ORB architectures, this research applies the following design process systematically.

1. Identify the ORB services whose behaviors may vary. This variation stems from which standard CORBA features are actually used and user's optional choice

for certain behavior. For example, CORBA provides an `Any` datatype that can store any other data-type. This feature is optional until used.

2. Apply the Virtual Component pattern [27] to make each ORB service pluggable, i.e., factor it out of the core ORB implementation.
3. Write concrete implementations for the different alternatives. For example, an implementation of the TCP/IP protocol or Secure Socket Layer (SSL) protocol. A factory [26], for example, a protocol factory, creates different protocols depending on configuration settings.

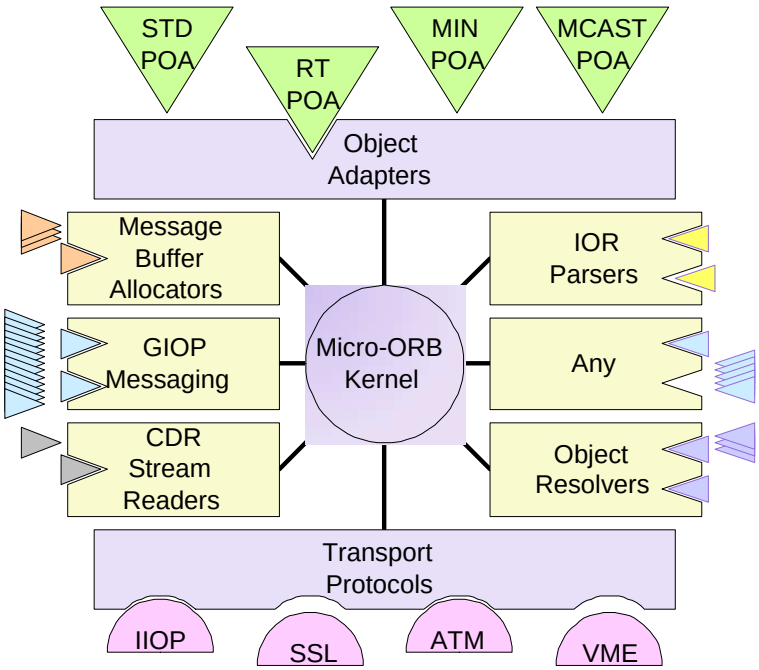

Figure III.2: Micro ORB Architecture

The systematic application of this design process to middleware results in a flexible and extensible *Micro ORB* design. The micro ORB design is based on the concept of *layered pluggability*, as shown in Figure III.2. In this architecture only a small ORB kernel is loaded in memory, with various components linked and loaded dynamically on demand. The advantage of this design is the significant reduction in footprint

and the increase in extensibility. In particular, independent ORB components can be configured dynamically to meet the needs of different applications. The remainder of this chapter uses a case study to demonstrate how a pluggable design based on the Virtual Component pattern, promotes customizability and simultaneously minimizes footprint.

Pluggable Middleware Component Design - A Cast Study

The POA is a standard component that enables programmers to compose servants portably across ORB implementations. Unlike its woefully underspecified predecessor—the basic object adapter (BOA)—the POA specification is well designed and provides standardized APIs for the POA operations. The following paragraphs summarize the important functionality provided by the POA:

1. Generating object references. The POA is responsible for generating object references for the CORBA objects it maintains. These references contain addressing information that allows remote clients to invoke operations on each object in a distributed system. This information is provided to the POA by the ORB Core and underlying operating system transport.

2. Behavior governed by policies. The POA provides a extensible mechanism for associating policies with servants in a POA. The values for policies are specified when a POA is created. Currently there are seven standard policies for the POA: *thread*, *lifespan*, *object id uniqueness*, *object id assignment*, *servant retention*, *request processing*, and *implicit activation*.

3. Activation and deactivation of objects. The creation of object references stem from the creation of a CORBA object. Once created, an object can alternate between *activated* and *deactivated* states. An object can service requests only if it is activated. The `deactivate_object()` operation is used to deactivate an object. It

is important to note that the lifetime of a CORBA object is different from that of the servant that implements it.

4. Incarnation and etherealization of servants. Servants implement CORBA object interfaces and can be registered with the POA implicitly or explicitly by application developers. To have requests delivered to it, however, an object must be *incarnated* by a servant. The POA can incarnate servants on demand. Etherealization of a servant breaks the association between it and its CORBA object.

5. Demultiplexing requests to servants. Client requests are sent as messages across the underlying OS transport. The POA demultiplexes these requests to the appropriate servants. The POA then invokes the appropriate operation on this servant.

6. SSI and DSI support. The POA allows programmers to construct skeletons that inherit from static skeleton interface (SSI) classes or a dynamic skeleton interface (DSI) class. Clients need not be aware that the request is serviced by a SSI or a DSI servant. Two CORBA objects supporting the same interface can be serviced, one by a DSI servant and one by a SSI servant. Further an object can be serviced by a DSI servant for some period of time, while being serviced by the static skeleton servant for the remaining time.

Portable Object Adapter Functionality and Architecture

The ORB is an abstraction visible to both the client and server. The POA, in contrast, is only necessary in a process performing the server role, so clients do not require the services of a POA. In ZEN, the ORB and the POA interact through a well-defined `ServerRequestHandler` interface. This design prevents the tight coupling of the ORB and the POA. This interface is specific to ZEN since the OMG has not standardized the interface between the ORB and the POA. In ZEN, the POA is only

one specific type of `ServerRequestHandler`. Variants, such as Real-time POA or Multicast POA, may handle requests and perform other POA activities, as well.

User-supplied servants are registered with the POA. Each client has an object reference, representing the remote servant, upon which it can invoke requests. When a request is made, it is passed as a message to the server. The POA then decides which servant the request corresponds to and invokes that operation on the appropriate servant. Figure III.3 shows the POA architecture. The architecture shown in this figure is implied by the interface to, and specification of, the POA in the CORBA specification. As long as an implementation of this architecture meets the designated CORBA semantics, however, it need not follow any prescribed design.

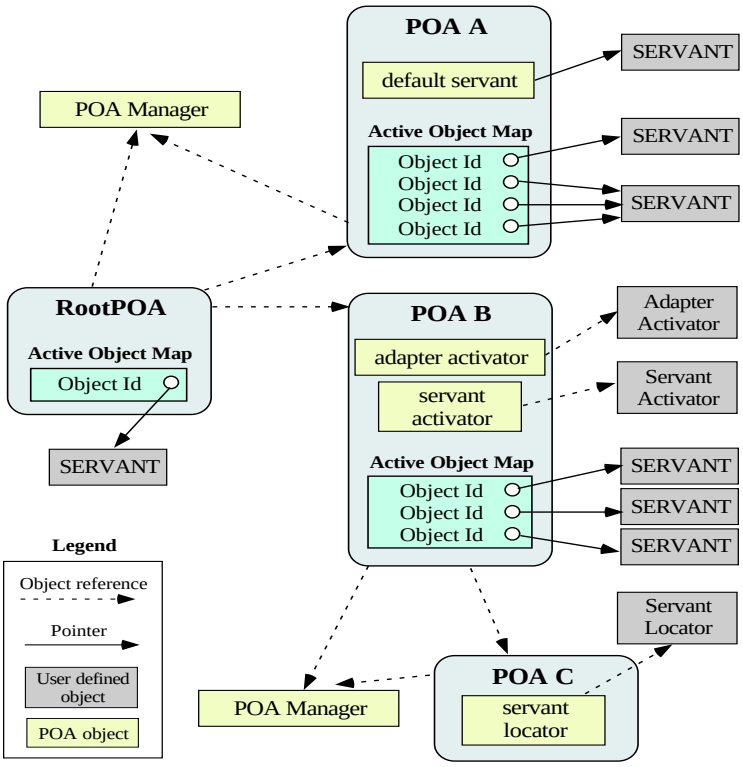

Figure III.3: The POA Architecture

Figure III.3 shows a special POA (called the `RootPOA`) that is always available to an application through the ORB factory method `resolve_initial_references()`.

Application developers can register servants with the root POA if the policies of the root POA specified in the POA specification are suitable for their applications.

A server application may establish multiple POAs to create a naming hierarchy (similar to the hierarchical directory structure found in an OS file system) that also allows setting of individual servant policies. For example, a server application might have two POAs:

- one supporting transient CORBA objects, whose lifetime can not exceed the POA in which it was activated, and
- the other supporting persistent CORBA objects, whose lifetime can exceed that of its activating POA.

Child POAs are created by invoking the `create_POA()` factory method on a parent POA. The server application in Figure III.3 contains three other nested POAs: `A`, `B`, and `C`. POA `A` and `B` are children of the root POA; POA `C` is `B`'s child. Each POA has an active object map that associates object ids to servants. Other key components in a POA are discussed below.

Adapter Activator. An adapter activator can be associated with a POA by an application. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not yet exist. The adapter activator can then decide whether or not to create the required POA on demand.

POA Manager. A POA manager encapsulates the processing state of one or more POAs. By invoking operations on a POA manager, server applications can cause requests for the associated POAs to be queued or discarded.

Servant Manager. A servant manager is a locality constrained servant that is provided by the application developer. The ORB uses a servant manager to activate and deactivate servants on demand. Servant managers are responsible for (1) managing the association of an object (as characterized by its Object Id value) with a particular servant and (2) for determining whether or not an object exists. There are

two types of servant managers: *ServantActivator* and *ServantLocator*. The type used in a particular situation depends on the policies in a POA, which are described in section /refPOAarch.

Alternate POA Designs

Having described the functionality of a POA, this section presents an overview of each of the three alternative POA design architectures that were implemented, measured, and compared: *monolithic POA*, *coarse-grain POA*, and *fine-grain POA*.

Monolithic POA Architecture

In a monolithic POA architecture, the POA is a single large component that contains the semantics needed to implement (1) policies in the OMG's POA specification and (2) ORB-specific policies. The monolithic design can increase the footprint (both code and data size) of the POA considerably since the POA implements the behavior required by the entire set of policies, rather than a minimal subset.

Monolithic POA also cannot be easily extended as new polices are added to the CORBA POA specification. Moreover, monolithic POA implementations complicate the addition of ORB-specific policies. Monolithic POA implementations also suffer from inefficiency in terms of redundant checking required to determine the appropriate course of action based on POA policies.

For example, during most operations a monolithic POA needs to check for the presence/absence of policies to incorporate the necessary behavior dictated by the policies associated with the POA. This overhead is incurred each time the operation is performed. For example, to process a client request, the POA must check for the request processing policy value associated with the POA, which then dictates how the request is processed.

Coarse-grain POA Architecture

In a coarse-grain POA architecture, the POA is still a single, large component. However by applying the Virtual Component pattern, the POA is treated as one component so it can be plugged-in or removed as shown in Figure III.4.

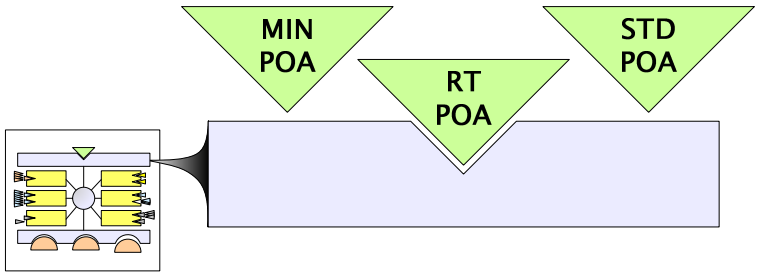

Figure III.4: Pluggable Object Adapters

A coarse-grain POA architecture is useful for pure clients, which need no object adapter and can reduce their footprint by completely removing all POA methods. It also useful for pure servers, which can reduce their footprint and achieve custom functionality by loading the most appropriate POA (*e.g.*, the `RootPOA` or a special group communication POA [51]) on demand. The coarse-grain pluggable POA design also simplifies the addition of new object adapters as they are standardized by the OMG. This coarse-grained POA architecture has been implemented in TAO [36] using the Component Configurator [88] pattern and dynamic link libraries (DLLs) to load each POA implementation variant.

Fine-grain POA Architecture

Aggressive use of the Virtual Component pattern allows greater subsetting of portions of the POA based on the application's needs. This architecture is called the "fine-grain POA architecture." In this approach, instead of an all-or-nothing loading of the POA, individual components of the POA can be loaded as needed. It is possible

to divide the POA into smaller pieces and make them virtual components. Such a fine-grain level of control can further reduce the footprint of a POA when it is needed by an application. It useful to decompose the POA as dictated by the values of the POA policies. Each of the CORBA POA policies has a set of policy values that specify the behavior of the POA with that policy. By breaking down the policies

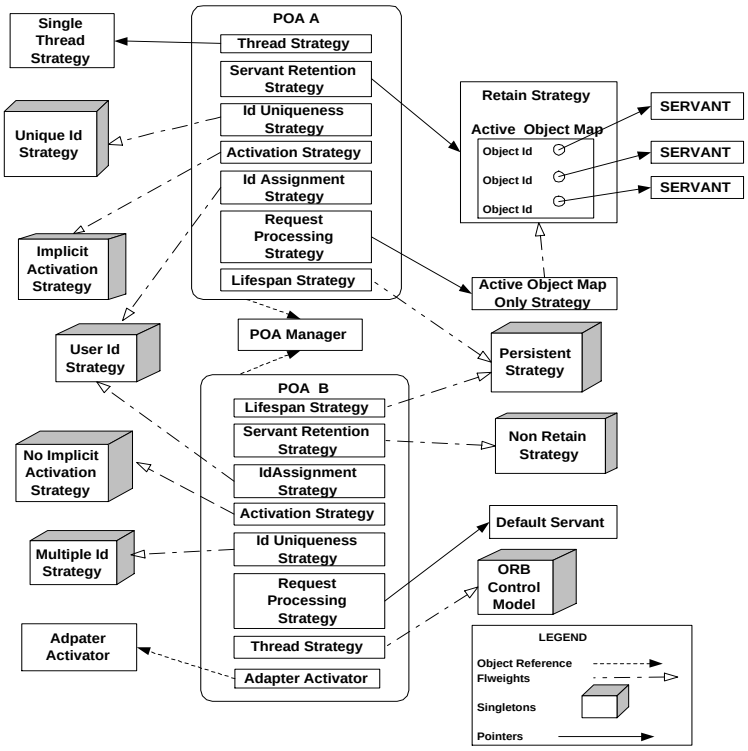

Figure III.5: Fine-grain Architecture of the ZEN POA

according to their possible values, it is therefore possible to load only the pieces that the POA needs, based on the list of policies specified at POA creation time. For example, Figure III.5 shows the fine-grain architecture of the ZEN POA, where each POA policy is factored out into a separate class hierarchy by applying the Strategy pattern [26], as described next.

Design of the ZEN Fine-grain POA

The remainder of this section describes how ZEN's POA is decomposed into modular components in accordance with POA policy values.

Primary POA Components

The four strategies described below are considered to be *primary components* in ZEN, *i.e.*, their behavior does not depend on other components. At POA creation time, these components are created first and hold the smallest amount of state. The following are the primary components of the POA.

ThreadStrategy component. This component implements the *Thread* policy, which is used to specify the threading model used in the POA. The POA can have one of the following threading models: *single thread*, *ORB controlled*, or *main thread*. If the POA is single-threaded, all the requests of the POA are processed sequentially. In a multi-threaded environment, all upcalls to the servant are invoked in a manner that is safe for multi-thread unaware code. In contrast, in the ORB-controlled model, multiple requests may be delivered simultaneously using multiple threads.

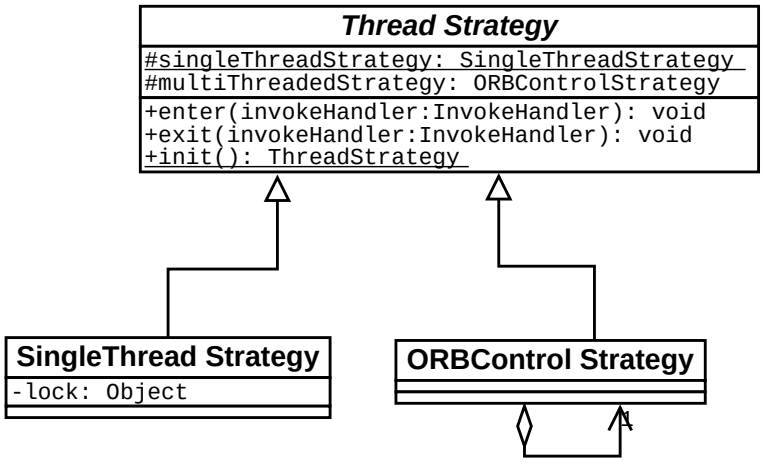

Figure III.6: ZEN's Thread Strategy

All requests to a main thread POA are processed sequentially in the thread that

runs the `main()` function. All upcalls made by POAs with this policy to servants are made in a manner that is safe for thread-unaware code. If the environment has special requirements that some code must run on a distinguished main thread, servant upcalls will be processed on that thread.

Using the Strategy pattern, the semantics of implementing the *Thread* policy can be strategized into two alternatives: class `SingleThread` and class `ORBControlModel`. Each class encapsulates the state and the logic of implementing the behavior specified by the policy. In ZEN,the main thread model strategy and the single thread strategy are equivalent. Figure III.6 shows the class diagram for ZEN's `ThreadPolicyStrategy` alternatives..

At POA creation time, a factory method `init()` in the base class `ThreadPolicyStrategy` creates the appropriate strategy instance based on the POA's policy list. Since the `ORBControl` component does not maintain state specific to a POA, it is implemented using the Flyweight pattern [26]. This pattern uses sharing to support large numbers of fine-grain objects efficiently, which means there is one instance of the `ORBControl` object. Each POA with that value for the `ThreadPolicyStrategy` policy will have a reference to that single object, thereby reducing memory usage.

Prior to making the upcall on the servant, the POA uses the `ThreadPolicyStrategy`'s `enter()` method. If the `SingleThread` strategy is in place, this method acquires a mutex lock. After the upcall is performed, the `exit()` method releases the lock. This synchronization is not present in the `ORBControlModel` strategy.

LifespanStrategy component. This component implements the *Lifespan* policy, which is used to specify whether the CORBA object references created within a POA are persistent or transient. Persistent object references can outlive the process in which they are created. Unlike persistent object references, transient object references cannot outlive the POA instance in which they were first created. After the POA is

deactivated, the use of object references generated from it will result in an `OBJECT_NOT_EXIST` exception.

The mechanism for implementing the POA's *Lifespan* policy has been separated into ZEN's `Persistent` and `Transient` strategies. Figure III.7 shows the class diagram of the `LifespanStrategy` component.

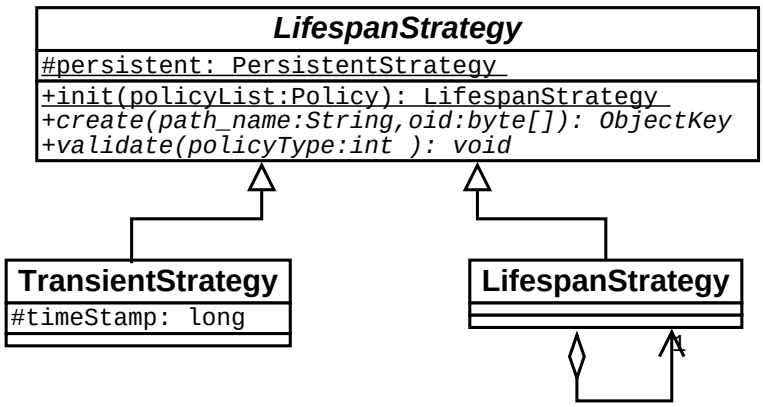

Figure III.7: ZEN's Lifespan Strategy

The responsibilities of the strategy include the creation of object ids for objects registered with the POA and validation of object keys contained in the client requests.

When asked to activate an object, the POA uses the `create()` method to generate an object id for the CORBA object. The object id generated depends on the concrete strategy loaded into the ORB. For example, the object id generated by a transient transient POA has a time stamp. When a client request is received, the `validate()` method of the `LifespanStrategy` determines whether it was this POA that generated the object id. If the POA is transient and the above is not true then a `OBJECT_NOT_EXIST` exception is returned to the client. In the persistent case, the adapter activator of the closest existing ancestor is used to create the POA automatically.

In ZEN, the persistent strategy does not maintain state specific to a POA, so it can be implemented as a flyweight `PersistentStrategy` object.

ActivationStrategy component. This component implements the `Activation` policy, which is used to specify whether implicit activation of servants is supported in the POA. If the implicit activation policy is active, it causes two things to happen when the servant method `_this()` is called:

1. The servant is registered with the POA and
2. The object reference for that servant is implicitly created.

Without this policy, the server must call either `activate_object()` or `activate_object_with_id()` to achieve this effect.

ZEN uses the `ActivationStrategy` shown in Figure III.8, to implement the behavior required by the *ImplicitActivation* policy.

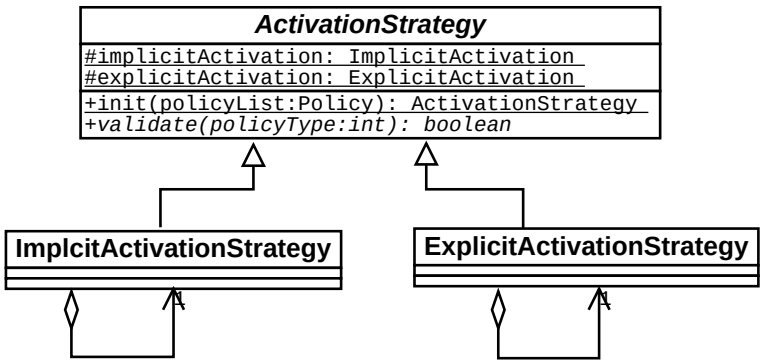

Figure III.8: ZEN's Activation Strategy

The `validate()` method is invoked to check if implicit activation is permitted, on this POA. Depending on the concrete strategy that is plugged into the ORB, the operation returns true or false. For example, the `servant_to_id()` and `servant_to_reference()` operations use the method to check if implicit activation is allowed.

Both of the following concrete strategies, `ImplicitActivationStrategy` and `ExplicitActivationStrategy` maintain no state within them and are implemented as flyweights to conserve memory.

Secondary POA Components

The secondary components in ZEN are strategies whose behavior depends on the values of primary strategies. These dependencies can lead to conflicts. When two policies cannot co-exist they are said to be in *conflict*. If the policy list specified at POA creation has conflicts, the strategies would also be in conflict. For example, if the *ImplicitActivation* policy value is `IMPLICIT_ACTIVATION`, the *IdAssignment* policy value cannot be `USER_ID`.

In ZEN, these conflicts are identified at strategy creation time (that is, *before* processing client requests), and appropriate response can be taken (for instance, raise an exception to the user, apply reflection to automatically select a non-conflicting set of policies, etc).

IdAssignmentStrategy component. This component implements the *IdAssignment* policy, which is used to specify whether object ids in the POA are generated by the application or by the POA. The possible object id assignment policy values are either `User-assigned` or `System-assigned`. Moreover, if the POA has both the SYSTEM_ID *IdAssignment* policy and PERSISTENT *Lifespan* policy enabled, object ids generated must be unique across all instantiations of the same POA. If the POA has the *ImplicitActivation* policy, this policy's value cannot be `USER_ID`. This subtle interaction between POA policy values is implicit, but must be enforced at POA creation time.

In ZEN, the `IdAssignmentStrategy` class models the behavior required by the Id Assignment policy. The interface of the `IdAssignmentStrategy` is shown in Figure III.9.

The `init()` factory method, that creates the concrete strategy also checks for conflicts and raises the `WRONG_POLICY` exception if necessary. The only responsibility of this strategy is to generate object ids for registering objects with the POA.

Under certain conditions, POA operations, such as `activate_object()` and `servant_`

`to_id()`, can activate servant using POA generated object ids. The `nextId()` method generates the new object id if the system id policy value is present in the POA. If the user id policy value is present, a `WRONG_POLICY` exception is raised. The semantics of incorporating the above behavior is present each of the concrete strategies.

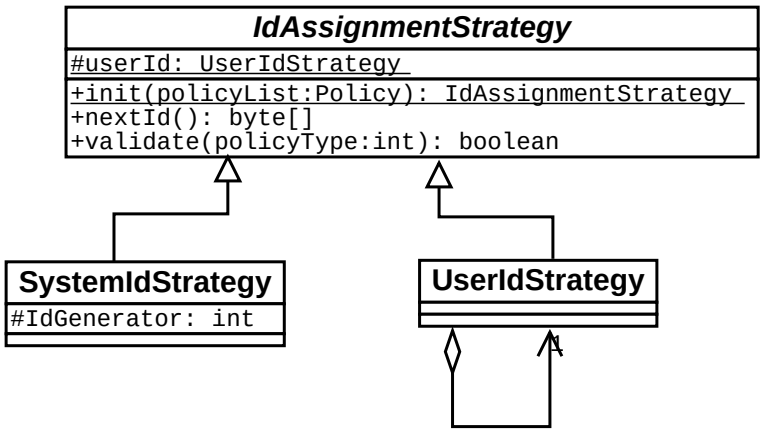

Figure III.9: ZEN's Id Assignment Strategy

The `UserIdStrategy` does not maintain any state specific to a POA, so it is designed as a flyweight.

IdUniquenessStrategy component. This component implements the *IdUniqueness* policy, which is used to specify if the servants activated in the POA must have unique object ids. If the policy value is unique id, servant activated by the POA support exactly one object Id. With the multiple id policy, servants activated by the POA may support multiple object Ids. The use of unique id policy value in conjunction with the *NonRetain* policy is meaningless. The OMG specification allows the ORB not to report an error if this combination is used, in ZEN this is considered to be in conflict and a `WRONG_POLICY` exception is raised.

The `IdUniquenessStrategy` enforces the behavior required by the policy. Figure III.10 shows the class diagram and the concrete strategies that extend the `IdUniquenessStrategy`. The `validate()`method is used by the POA to check for the policy

value associated with the POA. For example, `activate_object()` operation before activation of an already existing servant, calls the `validate()` method to check if

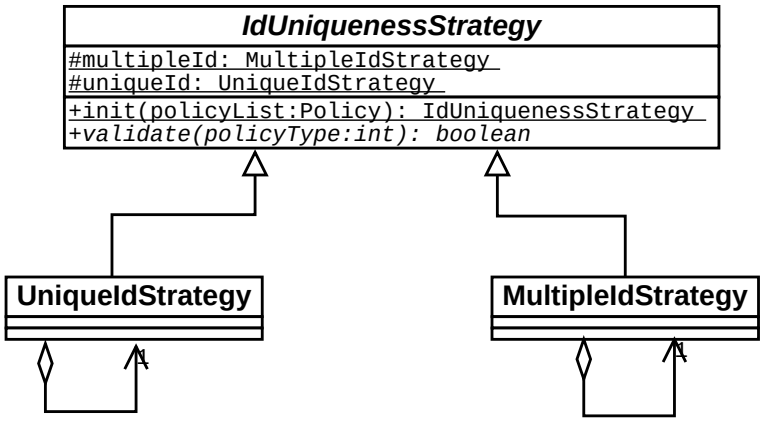

Figure III.10: ZEN's Id-Uniqueness Strategy

re-registration is permitted. Both the concrete strategies do not maintain any state within them and hence are designed as flyweight references.

ServantRetentionStrategy component. This component implements the *Servant Retention* policy, which is used to specify if the POA retains the active servants in an active object map. This policy can either have retain or non-retain as the possible policy values. Some combinations of POA policies are not allowed. For example, the *ServantRetention* policy may have a value of `NON_RETAIN` and an *ImplicitActivation* policy may have a value of `IMPLICIT_ACTIVATION`, but they cannot have those values simultaneously since they conflict with one another. Again, these implicit and subtle issues must be enforced at POA creation time.

In ZEN, the `ServantRetentionStrategy` models the behavior required by the *ServantRetention* policy. Figure III.11 shows the concrete strategies that extend the `ServantRetentionStrategy`.

The `ServantRetentionStrategy` maintains an active object map where the association between the CORBA object and the servant is maintained. If a POA has

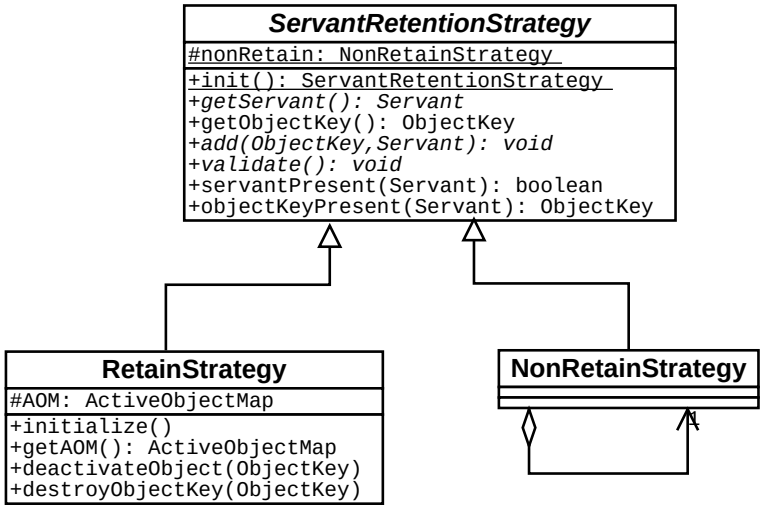

Figure III.11: ZEN's Servant Retention Strategy

unique id and retain policies, there exists a one-to-one relationship between the object ids and servants and vice versa.

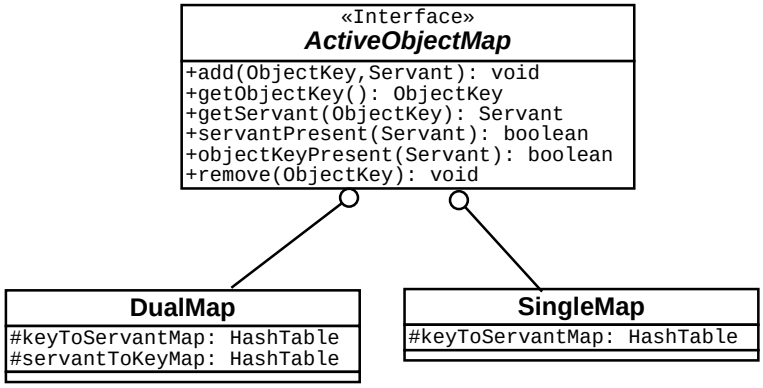

Figure III.12: ZEN's Active Object Map Interface

In this case, operations `servant_to_id()` and `servant_to_reference()` support reverse lookups (*e.g.*, given a pointer to a servant, return the object associated with it). To speed up these operations, ZEN uses a reverse map that maps servants to their object ids. Since this reverse map is only needed in certain cases, the active object map is further strategized into `SingleMap` and `DualMap`. Figure III.12 shows the active object map interface.

The optimization describe above further reduces the footprint of the POA when a multiple id policy value is used. In the traditional approach, operations requiring lookups on the active object map would have to be preceded by guard conditions that check if the POA has the retain policy. In ZEN, depending on the concrete strategy in place, these either produce the desired behavior or raise the `WRONG_POLICY` exception.

The `NonRetainStrategy` encapsulates the mechanism of enforcing the non-retain policy. This strategy does not maintain any state specific to the POA and is implemented as a flyweight. All POA's having the non-retain policy have references to this flyweight.

RequestProcessingStrategy component. This component implements the *RequestProcessing* policy, which specifies how the POA should process requests. On receipt of a request, the POA based on the request processing policy value can do one of the following.

- **Consult the active object map only.** The POA using the object id searches the map for the associated Servant. It then uses that servant to process the request. If unsuccessful, an exception is returned to the client.
- **Use a default servant.** If the POA has the *Retain* policy and Step 1 is unsuccessful, then a default servant if present is used to service the request. If a default servant has not been associated or the POA does not have the policy an exception is returned to the client.
- **Use Servant Manager.** If the POA has the *UseServantManager* policy, the application supplied manager can be asked to incarnate/activate a servant for the object id. This servant is used by the POA to service the request. Depending on the *ServantRetention* policy, the servant manager can either be a servant activator or a servant locator.

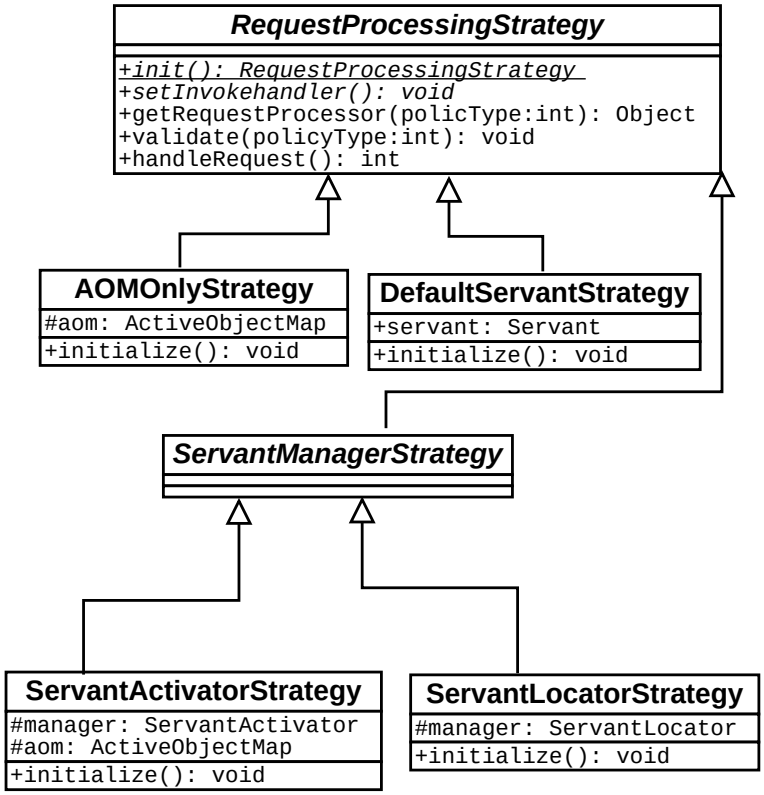

Figure III.13: Request Processing Strategy

The *RequestProcessing* policy is strategized along the three alternate courses of action mentioned above. Figure III.13 shows the class diagram for the `RequestProcessingStrategy`.

`ActiveObjectMapOnlyStrategy` encapsulates the logic of request dispatch if the active object map only policy is used. The POA uses the `handleRequest()` method of the base strategy strategy to service request.

The `DefaultServantStrategy` is associated with the POA if the appropriate policy value is used. Depending on the servant retention policy value, this strategy either consults the active object map first for request dispatch, or uses the default servant. If the non-retain policy value is used the POA, the servant is directly used. In either of the cases, if no servant is associated with the POA, an exception is raised.

The `ServantManagerStrategy` is associated with the POA if the Use Servant Manager policy value is specified. Moreover, depending on the *ServantRetention* policy for the POA, this is strategized into a `ServantActivatorStrategy` or a `ServantLocatorStrategy`. Each of these concrete strategies have the semantics necessary for request dispatch.

In a traditional POA implementation, each time a POA receives a request it must check the value of the request processing policy. In ZEN, however, the semantics of request processing in each case is present in the concrete strategy for that policy, so the policy value need not be checked at all.

Empirical Results

This section presents the results of blackbox benchmark measurements that compares the pluggable architecture of the ZEN POA with a monolithic POA architecture. All the experiments in this section 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. For these experiments, ZEN version 1.0 and JacORB [9] version 2.2 was executed on a Java SDK JVM version 1.4.2. To eliminate differences in the POA configurations, the following properties were set in both ZEN and JacORB:

1. Logging was turned off
2. POA monitoring was turned off for JacORB in the properties file.
3. The number of threads in the thread pool was set to 10
4. Maximum queue size was set to 100 and
5. No priority was set for the threads doing the request processing.
6. Interceptors were turned off
7. Servants are normal CORBA servants that inherit from `org.omg.PortableServer.Servant`, *i.e.*, DII and DSI were not considered

Root POA metrics

As discussed earlier in this section, the root POA is an integral part of every CORBA server and is always present, whether or not any other child POAs exist. A root POA suffices for many applications, unless the server needs to provide different QoS guarantees, such as object reference persistence. Thus, minimizing the footprint of the root POA is vital to minimizing server footprint. This test measures the increase in footprint after the root POA has been associated with the ORB. The memory increase prior to and after the call to the `resolve_initial_references()` gives the foot print increase contributed by the root POA.

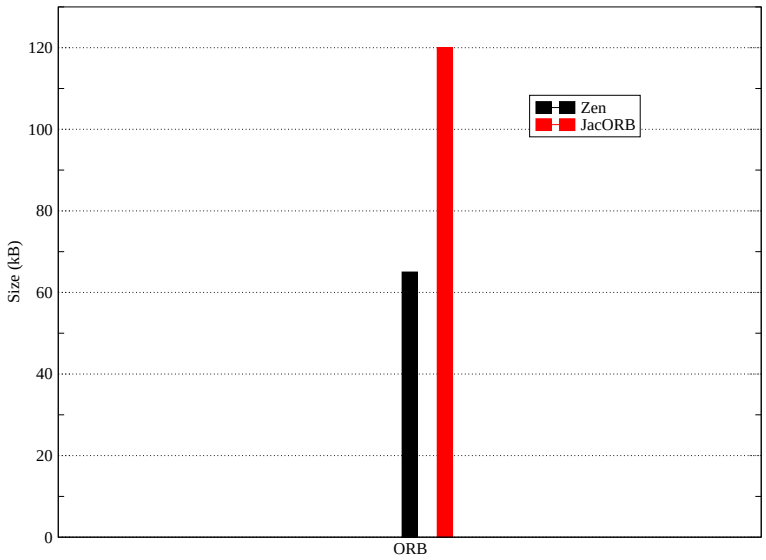

Figure III.14: Root POA Footprint

Figure III.14 illustrates that the footprint of the Root POA in ZEN is 65 kilo-Bytes, while that of JacORB is 120 kiloBytes. Thus, ZEN's root POA is almost half the size of JacORB's Root POA. In ZEN, the creation of the root POA results in initialization of all the base abstract strategies and the creation of the appropriate concrete strategies for the root POA policies. The root POA configuration maintains the maximum state among all POAs in ZEN. This small footprint bolsters the micro

POA design in the ZEN. Since JacORB is designed monolithically, it suffers from a larger memory footprint. Further, the creation time for ZEN RootPOA was ∼250 (μsecs) while JacORB incurred an higher latency of ∼375 (μsecs).

Child POA footprint metrics

A CORBA server creates child POAs for the CORBA objects if the QoS parameters differ from those provided by the Root POA. For example, an application developer creates a Child POA to provide object reference persistence or to associate multiple objects with a single servant, etc. This test measures the variation of footprint with the number of POAs created. The increase in footprint prior to and after the call to the `create_POA()` method is measured in each of the case. The following two scenarios were measured:

A The child POAs have the same policies as the root POA

B The child POAs have the following policy values: (1) *NonRetain* policy, (2) *ServantManager* policy, and (3) *UserId* policy.

The configuration (A) in ZEN, holds the maximum state while (B) corresponds to the least state. These configurations thus represent the minimum and maximum foot-print range for ZEN.

Figure III.15 shows how memory increases with the number of child POAs. For both cases, JacORB and ZEN, grow linearly. However, the growth of ZEN is more gradual. For example, in Configuration (A), for 32 POAs, JacORB's footprint is ∼650 kiloBytes while that of ZEN is ∼175 kiloBytes and for Configuration (B) JacORB footprint is ∼675 kiloBytes while ZEN incurs ∼120 kiloByes. For configuration A, the average POA size for ZEN is ∼7 kiloBytes where as JacORB's child POA is ∼15 kiloBytes, a factor of two difference. For configuration B, the average POA size for ZEN is ∼4.5 kiloByes while for JacORB it is ∼20.5 kiloBytes. Configuration B in ZEN corresponds to the scenario where maximum flyweight [26] references are used,

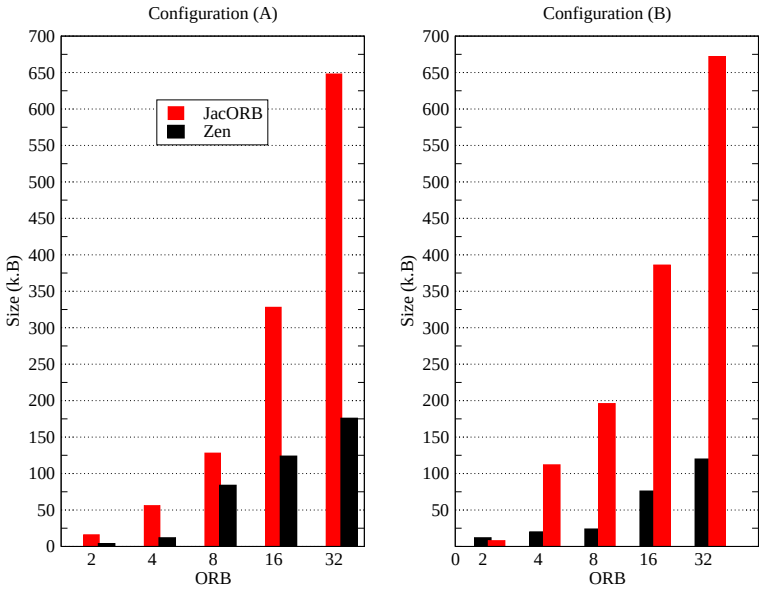

Figure III.15: Child POA Footprint Results

i.e., state is shared rather than duplicated for each POA. As shown in the results, this provides a factor of 5 improvement in footprint. Configuration A on the other hand represents the case where least amount of state is shared which limits the footprint improvements to a factor of 2. The results are again in agreement with the RootPOA results as both represent the same configuration.

Child POA creation time metrics

The design of the ZEN POA is expected to reduce the creation time of the POA since most of the concrete strategies are implemented as flyweights. In many applications, a POA could also be created as a side-effect of an upcall on the servant. In this scenario, a slow creation time for a child POA could decrease throughput and increase jitter. This test measures the variation in creation time with the number of POAs created. The child POAs that are created have the same policies as the root POA. In ZEN, default policy values (which are those used by the root POA) require more memory and are more expensive to create than are non-default policy values,

so this test exercises the worst-case scenario. Figure III.16 shows that both ZEN and

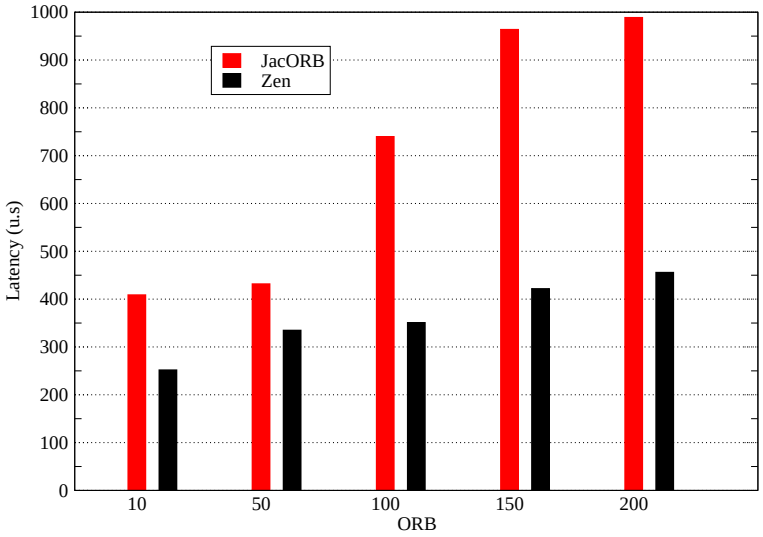

Figure III.16: Child POA Creation Time

JacORB grow linearly with the increase in the number of child POAs. However, the rate of increase for ZEN is very gradual and marginal than that of JacORB. From the samples, the average time for POA creation in ZEN is ∼3.5 μsecs while that for JacORB is ∼7.2 μsecs. On a average, therefore, JacORB is twice slower than ZEN for child POA creation.

Cost in memory per object activation

One key function of the POA is to generate object references. Thus, the cost in memory per object activation provides an indication of footprint increase as servants are associated with the POA. In this test, a servant is activated multiple times. Each time the servant is activated the POA creates an association between the servant with its object id in the active object map. The increase in footprint prior to and after the activation of servants is measured for each case. For this experiment both the ZEN

and JacORB POAs were associated with Id assignment policy of MULTIPLE_ID and Id assignment policy of SYSTEM_ID.

Figure III.17 illustrates that memory increases linearly with an increase in the number of activations. The increase of ZEN is slower than that of JacORB. For ZEN, on an average, the cost of memory per object activation is around ∼176 bytes while that of JacORB is ∼260 bytes.

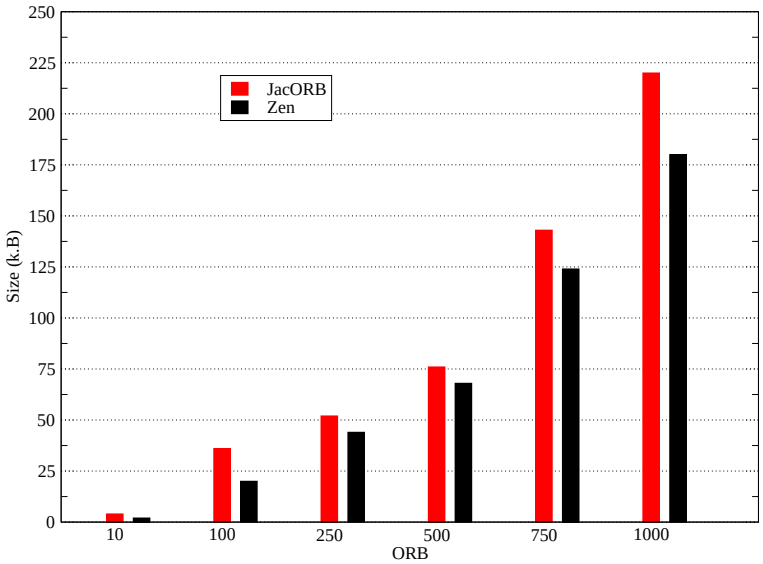

Figure III.17: Cost in Memory per Object Activation

For both ZEN and JacORB the contributing factors include creation of object keys and tables to store the association between the object keys and servants. The strategization of the POA in ZEN ensures that the object keys hold only necessary state.

Summary

This chapter showed how aggressive use of design patterns help in addressing QoS concerns of the PLA based DRE systems. Using the POA as an example, this chapter illustrated how *micro ORB* designs:

- **minimize footprint.** ZEN's POA design achieves a small memory footprint for middleware suitable for DRE systems. Each application incorporates only the sections of middleware code that it actually needs. By decomposing the POA into Virtual Components, the POA requires minimal memory for each application using ZEN.
- **Facilitate ease of adaptation.** A pluggable, highly-modular POA design applies the core software engineering concept of *separation of concerns*. Each of the Virtual Components in ZEN's POA encapsulates the implementation for a particular POA policy. This enables ZEN to associate custom POA policies, e.g. new algorithms, data structures, and capabilities.
- **Facilitate ease of configurability.** A pluggable, highly-modular POA design allows other custom policies to be associated with minimal cost. This allows the POA to be tailored according to the application domain.

CHAPTER IV

TECHNIQUES FOR SPECIALIZATING PLA MIDDLEWARE

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 paper extends earlier research on general-purpose middleware optimizations [69, 70, 72] by describing the results of developing and applying a toolkit to help resolve key aspects of the generality/specificity tension between general-purpose standards based middleware and application specific product variants in a PLA. This toolkit automates the *specialization* [16] of general-purpose, standards-based middleware to meet the needs of specific PLA-based DRE systems. In particular, this chapter provides the following research contributions:

1. It uses a representative PLA case study to identify key dimensions of *excessive generality* in standards-based middleware, focusing on Real-time CORBA [61], which is standard middleware used in Boeing Bold Stroke [75, 89, 91], which is a PLA-based DRE system in the domain of avionics mission computing.
2. It shows how *context-specific specialization techniques* [33] (such as code refactoring [24], and code weaving [105]) can be used to customize the widely used TAO [87] 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. It describes the design of a domain-specific language, tools, and a process for *automating the specialization techniques* discussed in the paper.
4. It presents and analyzes *quantitative results* that demonstrate the improvement in performance and predictability of specializations applied to TAO in the context of the PLA case study.

Middleware Specialization Challenges

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

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 the specialization approach. The scenario is based on the Boeing Bold Stroke avionics mission computing PLA [91], 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 IV.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.

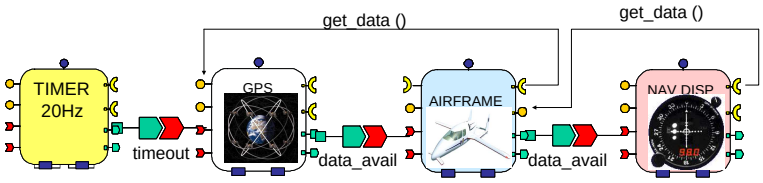

Figure IV.1: *BasicSP* Application Scenario

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.

Common Types of Excessive Generality in Middleware

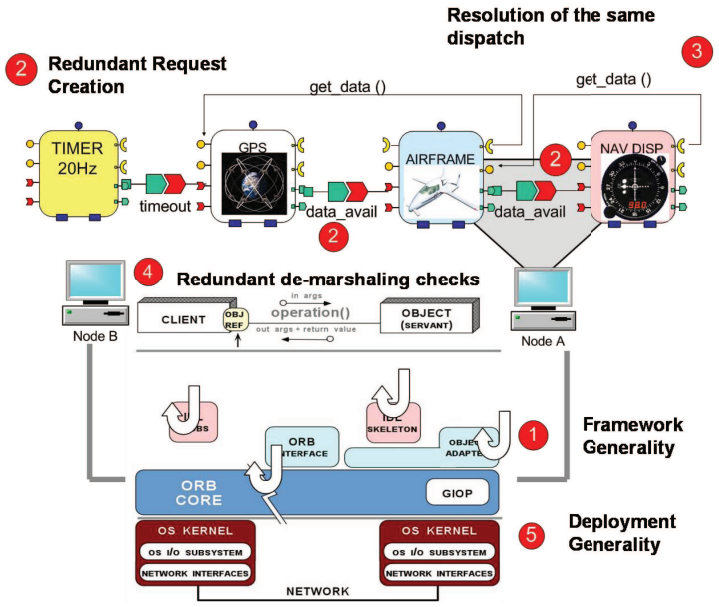

Figure IV.2: *BasicSP* Specialization Points

This section identifies and describe key types of excessive middleware generality that are relevant to PLA-based DRE systems. The *BasicSP* scenario from Figure IV.1 is used to show how this generality manifests itself in a PLA-based DRE system. The challenges of each type of generality are shown in Figure IV.2 and discussed below. These challenges are not limited to Real-time CORBA or the *BasicSP* PLA, which are simply used as examples to make the discussion concrete.

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 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.*

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 [69] 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 [69] and perfect-hashing [69] 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 upto 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.*

Applicability of Middleware Generality Challenges

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 [26, 88] 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.

Resolving Middleware Generality Challenges

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 [39]. Context-specific specializations can be realized using *code-refactoring and weaving* [24,105], which uses aspect-oriented programming mechanisms to factor out and weave crosscutting concerns, as well as *language mechanisms*, such as program optimization techniques [37]. This section, next describes how context-specific specializations have been applied to TAO to resolve the challenges described in Section IV. The specialization techniques and tools are reusable and applicable to various middleware implementations beyond TAO.

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* [52], 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. The program specialization strategies push invariant data through the middleware code, simplifying along the way. For example, the techniques 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. This section describes the *intent* (purpose), *invariance assumptions* (conditions in the *BasicSP* case study that enabled certain specializations), and *type* (technique) of specialization that have been applied to resolve the middleware generality challenges described in Section IV.

To evaluate the middleware specializations in a realistic context, they have been applied them to the TAO implementation of Real-time CORBA, which is written in C++ and contains many general-purpose ORB optimizations [70, 72]. General-purpose optimized TAO is used as a baseline to quantify the benefits of specializations that help resolve the challenges for PLAs described in Section IV. The techniques 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`).

Specialize Middleware Framework Extensibility via Aspect Weaving

This specialization technique resolves challenge 1 in Section IV.

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 this case study, the following TAO components are specialized (1) Reactor framework [85], which is responsible for demultiplexing connection and data events to their corresponding GIOP event handlers, and (2) pluggable protocol framework [64] 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 IV.3(A) shows different Reactor implementations supported by TAO.

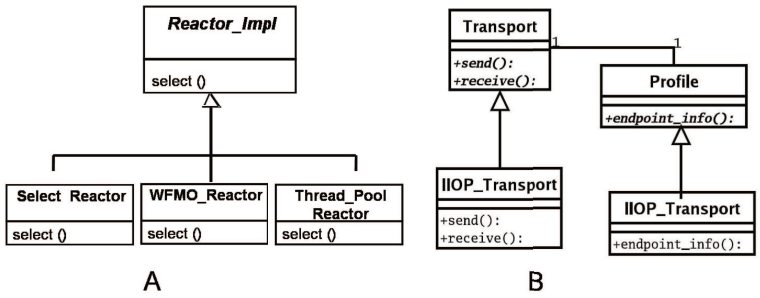

Figure IV.3: Reactor & Protocol Specialization

The `Select_Reactor` uses the single-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 [26] to configure different protocol implementations during the ORB initialization phase. As shown in Figure IV.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, the *BasicSP* PLA scenario only uses the `Select_Reactor` and VME protocol, so there is no need to incur additional indirection and generality overhead.

Specialize Request Creation/Initialization via Memoization

This specialization technique resolves challenge 2 described in Section IV.

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 IV.4 shows the structure of a two-way CORBA request using GIOP version 1.2.

As shown in the figure, every request has three components defined by the CORBA

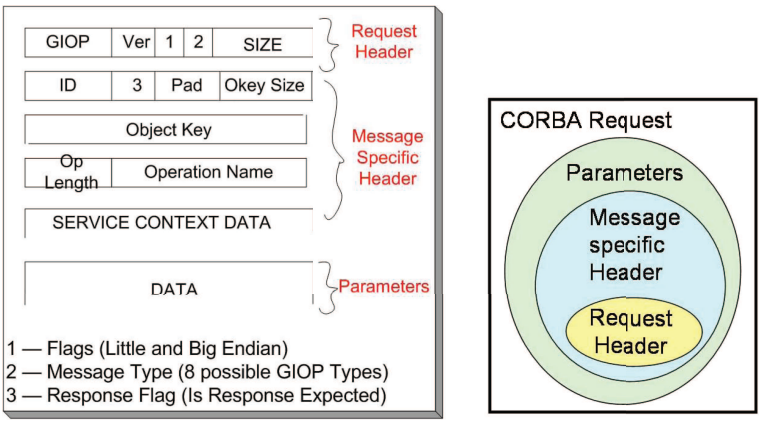

Figure IV.4: Opportunities for Request Creation Specialization

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 IV.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 the *BasicSP* PLA case study, the precomputed "result" is the CORBA request. This specialization thus avoids unnecessary creation and/or initialization of requests.

Specialize Dispatch Resolution via Layer-Folding

This specialization technique resolves challenge 3 described in Section IV.

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 IV.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 than navigating this layered path, a specialized implementation

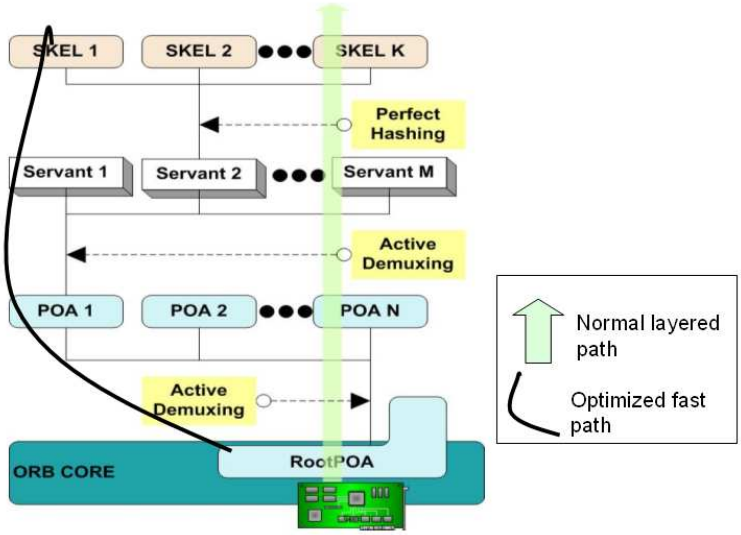

Figure IV.5: Specializing Request Dispatching

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 memo-ization, where an answer (in this case the dispatcher) is saved for later use than recomputing it each time. This enables multiple middleware layers to be collapsed during request processing.

Specialize Request Demarshaling via Constant Propagation

This specialization techniques resolves challenge 4 described in Section IV.

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 IV.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.

Specialize Platform Generality via Autoconf Mechanisms

This specialization technique resolves challenge 5 described in Section IV.

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. GNU's `autoconf` (`www.gnu.org/software/autoconf`) toolkit is used 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. TAO is extended 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. TAO was extended 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, GNU `autoconf` was used 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.

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 IV into such large code bases will clearly not scale. To augment GNU `autoconf`, this research has 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 developed to automate the specialization of middleware for PLA-based DRE systems. FOCUS is available at `www.dre.vanderbilt.edu/~arvindk/FOCUS`.

FOCUS Requirements and Goals

The 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 this approach by applying it to TAO. The types of specializations discussed in Section IV 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 [15] that combine annotations (directives) with code templates [74] 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.

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.

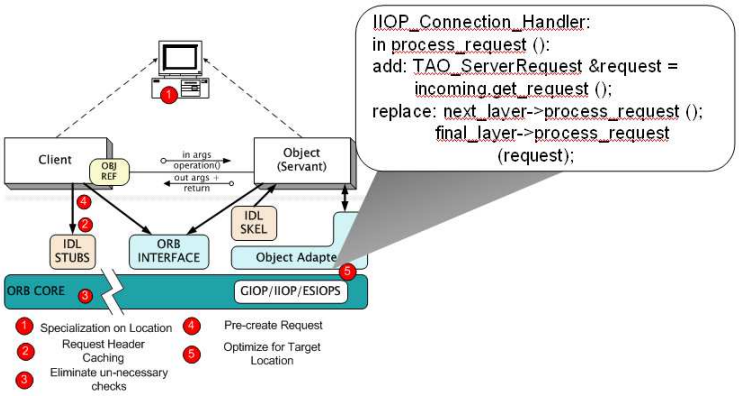

Figure IV.6: Capturing Specialization Transformation as Rules

Phase 1: Capturing specialization transformations. In this phase, as shown in Figure IV.6 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 IV illustrates portions of the transformations required to automate aspect weaving specializations in TAO.

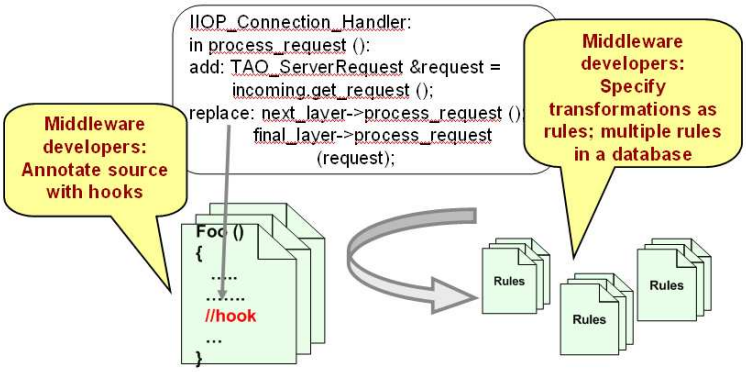

Figure IV.7: Annotating Middleware Source Code

Phase 2: Middleware annotation. In this phase, as shown in Figure IV.7, 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 IV 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 IV.8.

First, they 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

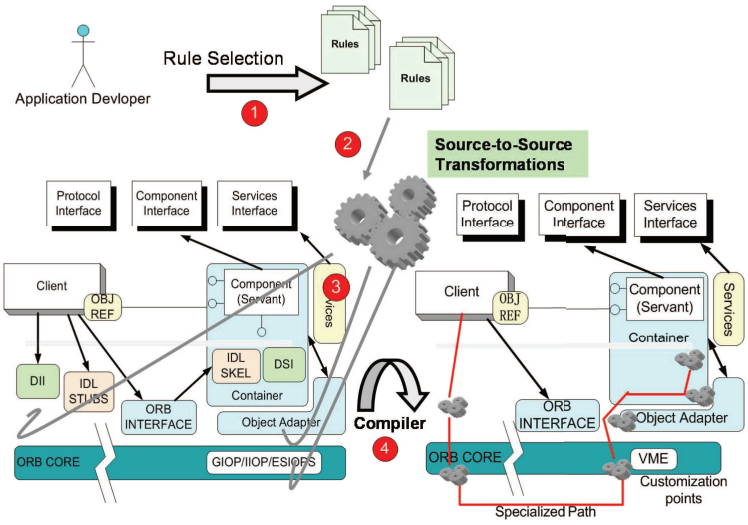

Figure IV.8: Steps in the FOCUS Transformation Process

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.

Discussion

This section described the specialization techniques, language, and tools developed to resolve middleware generality challenges in

Section IV. Table IV.1 lists the specialization techniques along with the corresponding FSL features we applied to resolve the generality challenges (Section IV

Table IV.1: Summary of 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

describes why FOCUS was not used to resolve (de)marshaling checks and deployment generality).

Similar to the challenges, the FOCUS 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. FSL tag names minimize the accidental complexity involved in specifying maintaining and adding specializations. Capabilities to capture specialization dependencies will be added as part of the future work on FOCUS. Ultimately, specialized middleware implementations will be synthesized from higher level PLA system models and functional specifications. FOCUS approach represents a step in this direction.

Applying Specializations to TAO – A Case Study

This section presents the results of applying the specialization tools described in Section IV 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 IV. The memoization, layer-folding, and aspect weaving were automated via the FOCUS toolkit described in Section IV.

Analyzing General-purpose Middleware

To specialize general-purpose Real-time CORBA middleware for PLA-based DRE systems, this section 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 IV. This code path is used to provide a baseline for comparing the context-specific specializations to quantify the number of steps specialized along the critical request/response processing path shown by the numbered bullets in Figure IV.9.

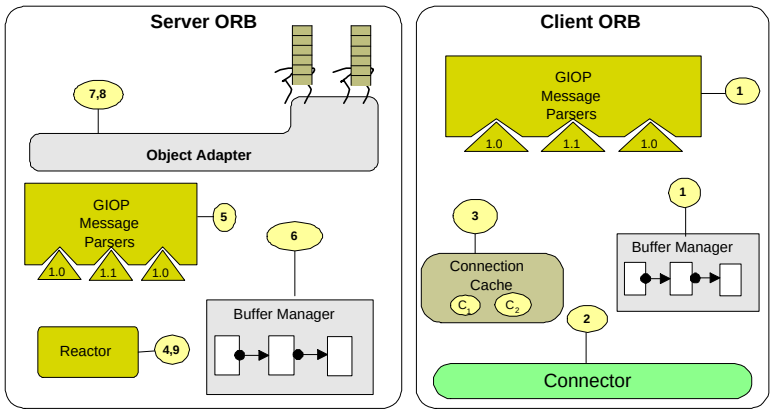

Figure IV.9: End-to-End Request Processing Path

Using this figure as a guide, the section now describe the steps involved 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:[1]

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) [71], 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.

[1]This discussion has been generalized using the Reactor, Acceptor-Connector, and Leader/Followers patterns [88], which are used in many popular CORBA ORBs, such as e*ORB, ORBacus, Orbix, and TAO.

Specializing TAO Middleware

Having outlined the activities at the client and server, this section now (1) describe how specializations have been applied to TAO using invariance assumptions to resolve the challenges for PLAs described in Section IV in the context of the Bold Stroke *BasicSP* scenario and (2) quantitatively compare the end-to-end latency, throughput, and predictability improvements accrued from this 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, the specializations largely comply with the Real-time CORBA specification and do not modify any standard interfaces or *BasicSP* application code.

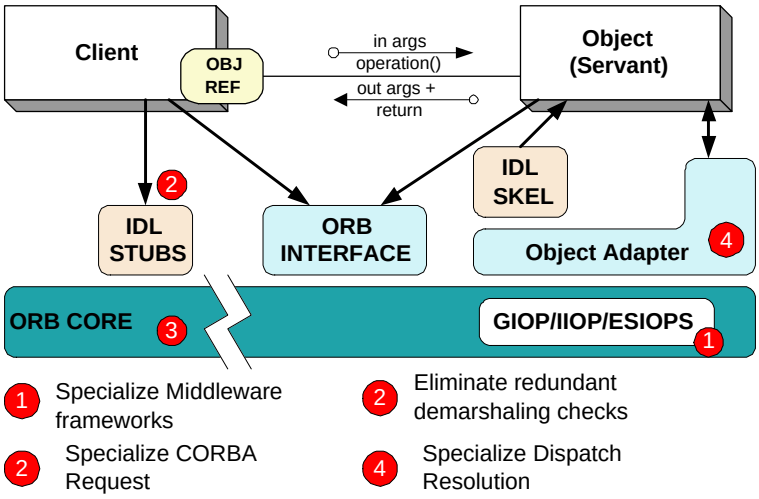

Figure IV.10: Specialization Points for TAO Real-time CORBA Middleware

Figure IV.10 illustrates where these specializations were applied to the Real-time

CORBA architecture. Our specializations 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, the following configuration was used: (1) portable interceptors are not used, (2) servants inherit statically from `org::omg::PortableServer::Servant`, *i.e.*, CORBA's dynamic invocation/skeleton features were not considered, (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 [75, 91].

To showcase the 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 IV through IV:

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, the specialization description illustrates (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 the experiments, *predictability* is defined as the measure of standard deviation of the data points.

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 IV.9.

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 [26], and (3) eliminating virtual methods from the reactor and interfaces within the middleware. To automate the specialization, FSL was used to capture the transformations, some of which are shown below.

```
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>
```

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 `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. Code snippet below shows how the middleware source code was annotated with hooks based on the FSL specialization file shown earlier.

```
//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;
}
```

Code snippet below illustrates how FOCUS transformed source code so that the base Reactor (`ACE_Reactor_Impl`) is replaced with the specialized Reactor (`ACE_Select_Reactor`).

```
//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:
}
```

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, <copy-from-source> capabilities provided by FSL was used.

Code snippet below shows how the concrete protocol specific implementations of template methods defined in the `Profile` class are copied from the `IIOP_Profile` class.

```
 <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>
```

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 this specialization was completed, support of IPv6 protocol was added to TAO, but our specializations required no changes.

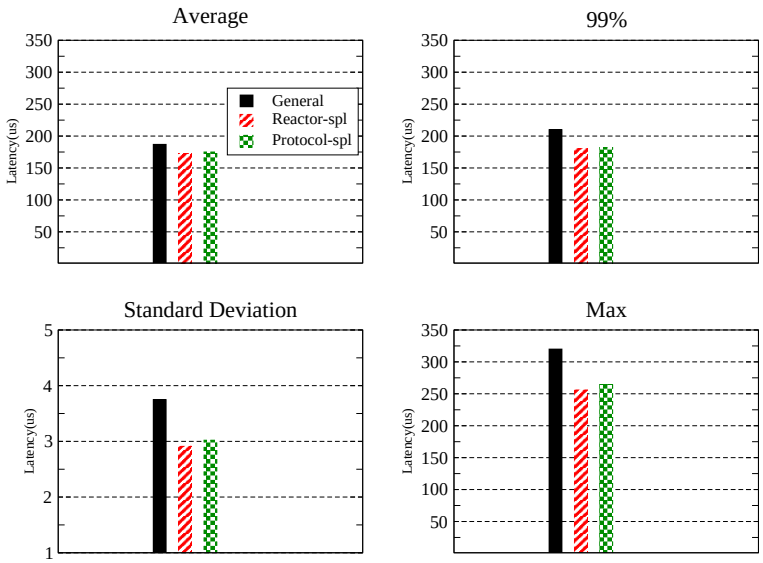

Figure IV.11: Results for Reactor & Protocol Specializations

Empirical results. Figure IV.11 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, the most representative end-to-end

results are presented. Our specialization results in average latency improvements of $\sim$8μsecs (4%) for the reactor case and in $\sim$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, future work in this area 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.

Applying the Memoization Specialization

This specialization corresponds to step 1 in the client side of Figure IV.9.

Specialization automation. In TAO, the GIOP engine creates protocol specific request/response objects. Listing below illustrates a portion of the transformations for this specialization.

```
<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>
```

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.

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 the invariance assumption from *BasicSP* scenario, where the `get_data` and `data_avail` operation are sent along separate connections.

Empirical results. Figure IV.12 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 $\sim 8\mu$sec (4%), while the path specialization results improved by 25%. This discrepancy shows how much the 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

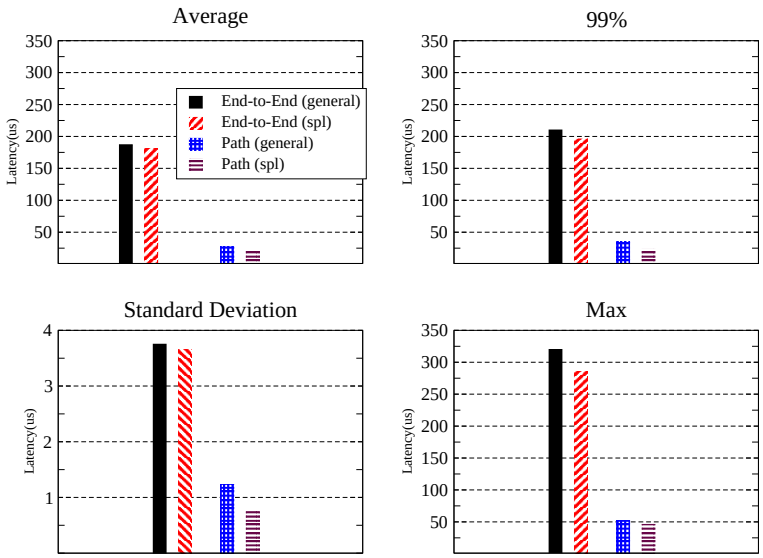

Figure IV.12: Results for Request Creation/Initialization Specialization

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.

Applying the Layer-Folding Specialization

This specialization corresponds to step 6 to step 8 in the server side of Figure IV.9.

Specialization automation. We implemented the layer-folding specialization (Section IV) 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 IV 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 the invariance assumption from the *BasicSP* scenario that the operations are same only on a per-connection basis.

Empirical results. Figure IV.13 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 $\sim$30μsecs, which is $\sim$16% better than the general-purpose TAO implementation. For the actual code path specialized this translates to $\sim$40% improvement in latency. The dispersion measures for end-to-end latencies improved by a factor of $\sim$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

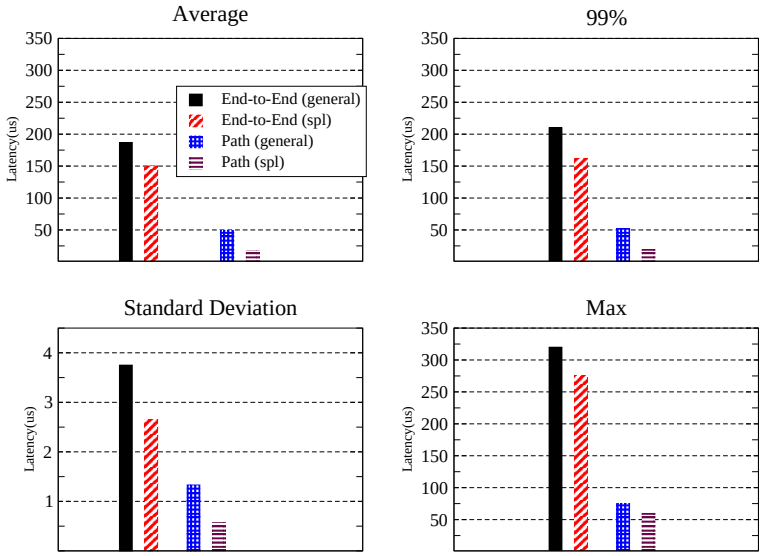

Figure IV.13: Results for Dispatch Resolution Specialization

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*.

Applying the Constant Propagation Specialization

This specialization corresponds to steps 1 and 11 on the client and steps 4 and 6 in the server of Figure IV.9.

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 IV.14 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 $\sim 8\mu$secs (a 4% improvement over the general-purpose TAO implementation), while eliminating byte order checks improved byte order checks by $\sim 9\mu$sec (a 4% improvement over the general-purpose TAO implementation). Path specialization results improved by $\sim 4 - 5\mu$sec (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, the specialization improvements were relatively small since the 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, another experiment was conducted 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

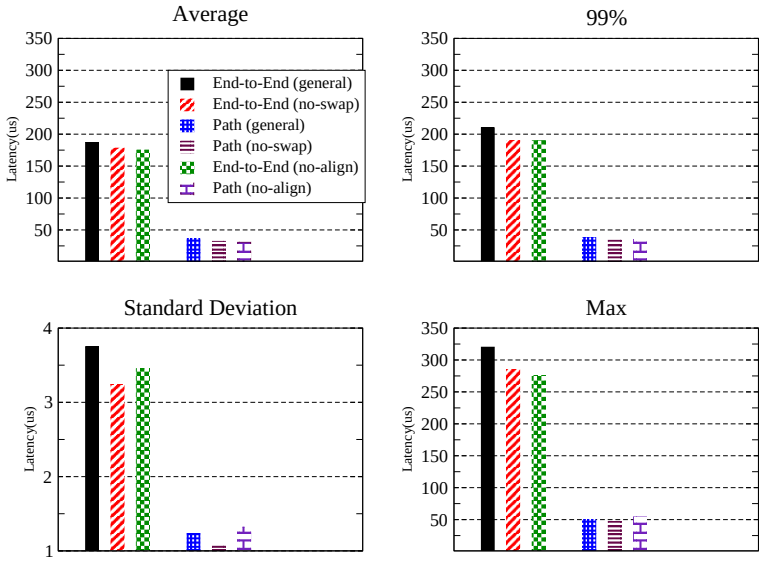

Figure IV.14: Results for Request (De)marshaling Specialization

TAO middleware to re-align the individual primitive types. The specialized TAO middleware, however, did not incur this overhead.

Table IV.2: Performance Speedup as a Function of Sequence Length

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

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 IV.2 illustrates the speed up in average end-to-end latency accrued from applying the 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.

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 the *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 the invariance assumption.

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

Empirical results. Figure IV.15 illustrates how applying the 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 the 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.

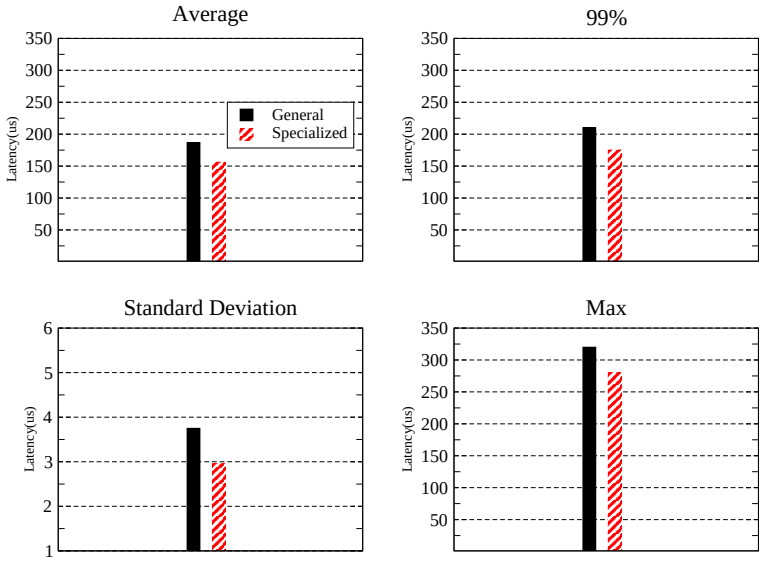

Figure IV.15: Results for Specializing Deployment Platform

Applying the Specializations Cumulatively

Figure IV.16 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.

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, the experiment ran using the complex data structure from the demarshaling experiments.

Table IV.3 shows that as the sequence length increases average end-to-end latency reduces considerably. For example, when the sequence length is 64 average latency improves by ∼26% while for length 4,096, latency measures improve by ∼51%.

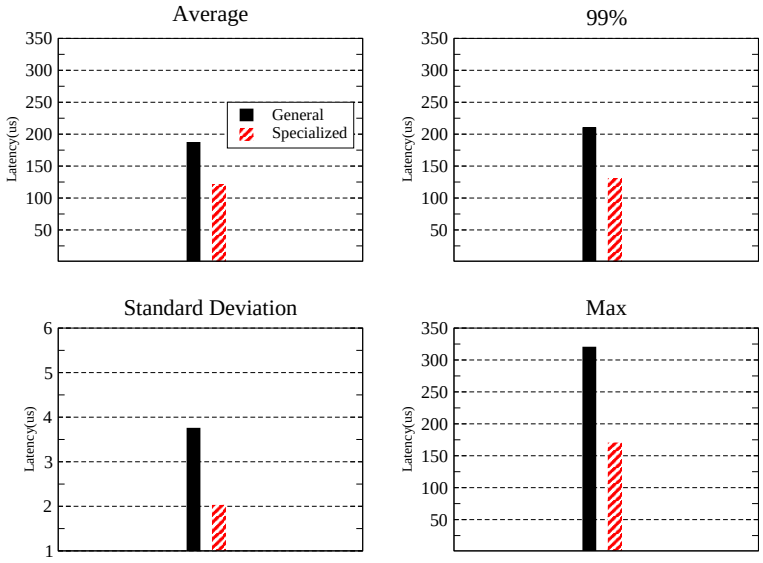

Figure IV.16: Results for Cumulative Specialization Application

Table IV.3: Cumulative Specialization Results as a Function of Sequence Length

Sequence Length	**Speedup**
64	26%
128	35%
1024	39%
2048	46%
4096	51%

Evaluating FOCUS

Having illustrated how FOCUS's DSL, tools, and process can be applied to help middleware developers build and evaluate middleware specializations, this section evaluates its benefits and drawbacks.

Benefits. In resolving the challenges described in Section IV, 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 IV 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 IV) 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.

Having validated the benefits of automating middleware specializations, the limitations presented above will be addressed in the future work on more powerful specialization languages.

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 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 the 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. 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 ∼65%, its average- and worst-case end-to-end latency measures by ∼43% and ∼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 [69, 70, 72]. 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 the specialization techniques and tools.

Implications on QoS. The specializations discussed in this paper had no interdependencies, *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 applicability to different middleware implementations. The magnitude of the QoS improvements from the specialization are specific to a particular implementation (TAO). Preliminary results [16] from applying context-specific specializations to ZEN (which is an open-source Real-time Java implementation of Real-time CORBA available at `www.zen.uci.edu`) were somewhat better than those shown in this paper. The difference stems in part from the fact that TAO is much more mature and optimized than ZEN, so there was less time/space overhead to optimize. To enable other middleware developers to analyze and implement the specializations the future work on this research will work on developing a comprehensive CORBA *specialization model* based on [16] that – independent of a particular CORBA implementation – identifies (1) points in an ORB architecture where specialization is beneficial and (2) API extensions to the architecture that provide *hooks* for achieving effective specialization.

Another observation is that the dispatch resolution optimization improved latency measures by 16% beyond active demultiplexing and perfect hashing general-purpose optimizations. These optimizations ensure $O(1)$ lookup time bound for all dispatch operations in TAO for the general case and incur the least overhead compared to other strategies [69]. Naturally, using other lookup techniques (such as linear search, which incurs $O(n)$ time bound) would yield an even greater payoff from the specializations.

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 the 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.

CHAPTER V

TECHNIQUES FOR VALIDATING PLA MIDDLEWARE CONFIGURATION TO ENSURE QOS

An inherent characteristic of high performance flexible and customizable middleware is (1) it runs on many hardware/OS platforms and interoperate with many versions of related software frameworks/tools and (2) provides support for end-to-end QoS properties, such as low latency and bounded jitter. These implementations have 10's - 100's of configuration options and customization parameters that PLA application developers can adjust to tailor the middleware to meet various functional and QoS needs. Specialized middleware (as described earlier) also influence end-to-end QoS of PLA applications.

Mapping product-line QoS requirements onto highly flexible middleware can be problematic, however, due in large part to the complexity associated with configuring and customizing QoS-enabled middleware. Time and resource constraints often limit developers to assessing the QoS of their DRE systems on very few configurations and extrapolating these to the much larger configuration space. In this context, the research challenges include (1) developing software processes to systematically and efficiently evaluate system QoS and (2) designing tools to synthesize necessary artifacts, such as benchmarking code to evaluate system QoS for various configuration options and (3) validating the general-purpose or specialized versions of middleware across different platforms. The remainder of this chapter discusses how these challenges have been addressed in OPTEML.

Model Driven Distributed Continuous QA Process

To specifically address the repeatability, cost and automation limitations of middleware configuration tuning approaches discussed in Section II, this research synergistically combines Model-Driven Development (MDD) techniques with Quality Assurance (QA) approaches. An MDD approach (for example the CoSMIC [28] project) resolves the accidental complexities involved in handcrafting scaffolding code such as XML-meta data, configuration files and benchmarking code for evaluating the QoS of middleware configurations across a range of product-line variants. Combining MDD approaches with Distributed Continuous Quality Assurance (DCQA) techniques [55] enables the validation of the different middleware configurations for functional correctness and performance across diverse platforms by executing QA tasks continuously and intelligently. In particular this process:

1. Minimizes the time and effort associated with testing various configuration options on particular platforms,
2. Provides a framework for seamless addition of new test configurations corresponding to various platform environment and application requirement contexts,
3. Facilitates automation, *i.e.*, continuously run tests that allow developers and end-users to evaluate the performance of software in various contexts and
4. Analyzes empirical data gathered and feedback results into modeling and decision making tools that systematically identify bottlenecks and optimize performance.

The remainder of this section provides a detailed description of the Model-driven DCQA environment and uses a concrete case study to show how this process can be applied to the *BasicSP* (described in Section IV) PLA scenario to quantify the impact of different middleware configurations on QoS.

Overview of Skoll DCQA Architecture

To address limitations with in-house QA approaches, the Skoll project is developing and empirically evaluating feedback-driven processes, methods, and supporting tools for *distributed continuous QA*. In this approach software quality is improved – iteratively, opportunistically, and efficiently – around-the-clock in multiple, geographically distributed locations. To support distributed continuous QA processes, the following set of components and services have been called the *Skoll infrastructure*, which includes languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and analysis techniques for characterizing faults.

The Skoll infrastructure performs its distributed QA tasks, such as testing, capturing usage patterns, and measuring system performance, on a grid of computing nodes. Skoll decomposes QA tasks into subtasks that perform part of a larger task. In the Skoll grid, computing nodes are machines provided by the core development group and volunteered by end-users. These nodes request work from a server when they wish to make themselves available. The remainder of this section describes the Skoll infrastructure and processes.

Skoll QA processes are based on a client/server model. Clients distributed throughout the Skoll grid request *job configurations* (implemented as QA subtask scripts) from a Skoll server. The server determines which subtasks to allocate, bundles up all necessary scripts and artifacts, and sends them to the client. The client executes the subtasks and returns the results to the server. The server analyzes the results, interprets them, and modifies the process as appropriate, which may trigger a new round of job configurations for subsequent clients running in the grid.

At a lower level, the Skoll QA process is more sophisticated. QA process designers must determine (1) how tasks will be decomposed into subtasks, (2) on what basis and in what order subtasks will be allocated to clients, (3) how subtasks will be

implemented to execute on a potentially wide set of client platforms, (4) how subtask results will be merged together and interpreted, (5) if and how should the process adapt on-the-fly based on incoming results, and (6) how the results of the overall process will be summarized and communicated to software developers. To support this process we've developed the following components and services for use by Skoll QA process designers (a comprehensive discussion appears in [55]):

Configuration space model. The cornerstone of Skoll is its formal model of a DCQA process' configuration space, which captures all valid configurations for QA subtasks. This information is used in planning the global QA process, for adapting the process dynamically, and aiding in analyzing and interpreting results.

Intelligent Steering Agent. A novel feature of Skoll is its use of an *Intelligent Steering Agent* (ISA) to control the global QA process by deciding which valid configuration to allocate to each incoming Skoll client request. The ISA treats configuration selection as an AI planning problem. For example, given the current state of the global process including the results of previous QA subtasks (*e.g.*, which configurations are wn to have failed tests), the configuration model, and metaheuristics (*e.g.*, nearest neighbor searching), the ISA will chose the next configuration such that process goals (*e.g.*, evaluate configurations in proportion to known usage distributions) will be met.

Adaptation strategies. As QA subtasks are performed by clients in the Skoll grid, their results are returned to the ISA, which can learn from the incoming results. For example, when some configurations prove to be faulty, the ISA can refocus resources on other unexplored parts of the configuration space. To support such dynamic behavior, Skoll QA process designers can develop customized *adaptation strategies* that monitor the global QA process state, analyze it, and use the information to modify future subtask assignments in ways that improve process performance.

Skoll in Action

At a high level, the Skoll process is carried out as shown in Figure V.1.

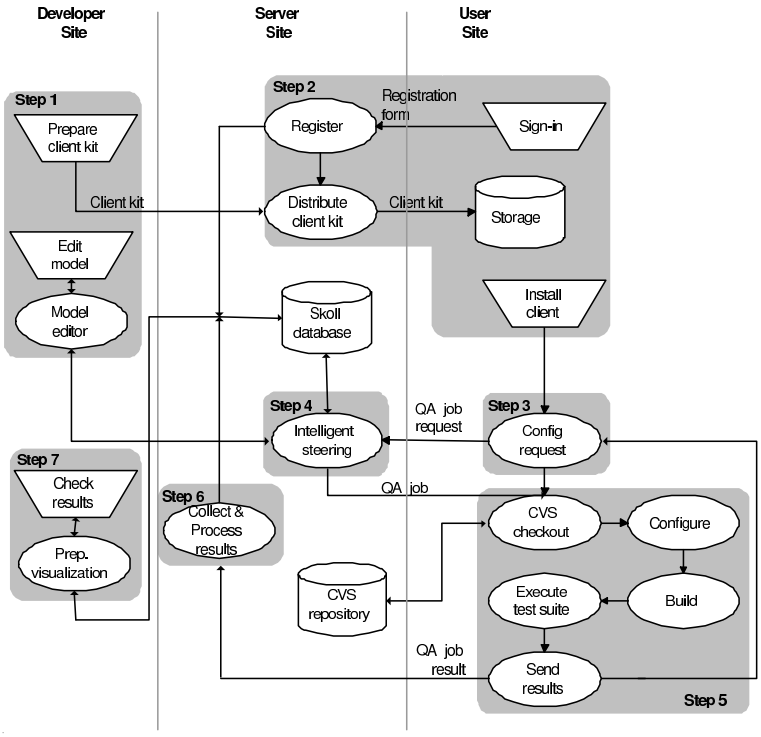

Figure V.1: Skoll QA Process View

1. Developers create the configuration model and adaptation strategies. The ISA automatically translates the model into planning operators. Developers create the generic QA subtask code that will be specialized when creating actual job configurations.

2. A *user* requests Skoll client software via the registration process described earlier. The user receives the Skoll client software and a configuration template. If a user wants to change certain configuration settings or constrain specific options he/she can do so by modifying the configuration template.

3. A Skoll client periodically (or on-demand) requests a job configuration from a Skoll server.

4. The Skoll server queries its databases and the user-provided configuration template to determine which configuration option settings are fixed for that user and which must be set by the ISA. It then packages this information as a planning goal and queries the ISA. The ISA generates a plan, creates the job configuration and returns it to the Skoll client.

5. A Skoll client invokes the job configuration and returns the results to the Skoll server.

6. The Skoll server examines these results and invokes all adaptation strategies. These update the ISA operators to adapt the global process.

7. The Skoll server prepares a *virtual scoreboard* that summarizes subtask results and the current state of the overall process. This scoreboard is updated periodically and/or when prompted by developers.

Overview of BGML MDD Tool

The initial Skoll prototype provided a distributed continuous QA infrastructure that performed functional testing, but did not address QoS issues, nor did it minimize the cost of implementing QA subtasks. In particular, integrating new application capabilities into the Skoll infrastructure (such as benchmarks that quantified various QoS properties) required developers to write test cases manually. Likewise, extending the configuration models (*e.g.*, adding new options) required the same tedious manual approach.

In the initial Skoll approach, creating a benchmarking experiment to measure QoS properties required QA engineers to write (1) the header files, source code, that implement the functionality, (2) the configuration and script files that tune the underlying ORB and automate running tests and output generation, and (3) project build files (*e.g.*, makefiles) required to generate the executable code. Initial feasibility studies [55] revealed how this process was tedious and error-prone. The remainder

of this section describes how the model-based techniques [95] have been applied in OPTEML to design the Benchmak Generation Modeling Language (BGML).

BGML provides visual representations for defining entities (components), their interactions (operations and events) and QoS metrics (latency, throughput and jitter). Further, the visual representations themselves are customizable for different domains. BGML has been tailored towards evaluating the QoS of implementations of the CORBA Component Model (CCM) [5][1].

BGML is built atop the Generic Modeling Environment (GME) [45], which provides a meta-programmable framework for creating domain-specific modeling languages and generative tools. GME is programmed via *meta-models* and *model interpreters*. The meta-models define modeling languages called paradigms that specify allowed modeling elements, their properties, and their relationships. Model interpreters associated with a paradigm can also be built to traverse the paradigm's modeling elements, performing analysis and generating code.

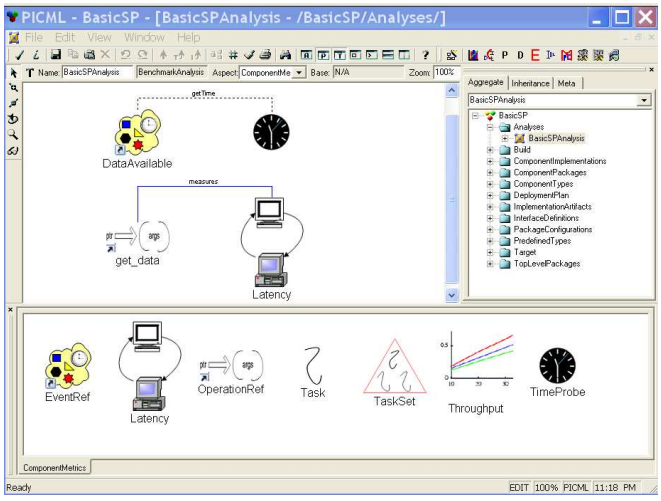

Figure V.2: QoS Validation via BGML

[1]BGML focuses on CCM since it is standard component middleware that is targeted for the QoS requirements of DRE systems. As QoS support for other component middleware matures BGML will be enhanced and applied to them as well.

BGML captures key QoS evaluation concerns of performance-intensive middleware. Middleware/application developers can use BGML to graphically model interaction scenarios of interest as shown in Figure V.2. Given such a model, BGML then generates most of the code needed to run experiments, including scripts that start daemon processes, launch components on various distributed system nodes, run the benchmarks, and analyze/display the results. BGML allows CCM users to:

1. Model interaction scenarios between CCM components using varied configuration options, *i.e.*, capture software variability in higher-level models rather than in lower-level source code.
2. Automate benchmarking code generation to systematically identify performance bottlenecks based on mixing and matching configurations.
3. Generate control scripts to distribute and execute the experiments to users around the world to monitor QoS performance behavior in a wide range of execution contexts.
4. Evaluate and compare CCM implementation performances in a highly automated way the overhead that CCM implementations impose above and beyond CORBA 2.x implementations based on the DOC model.
5. Enable comparison of CCM implementations using key metrics, such as throughput, latency, jitter, and other QoS criteria.

With BGML, QA engineers graphically model possible interaction scenarios. Given a model, BGML generates the scaffolding code needed to run the experiments. This typically includes Perl scripts that start daemon processes, spawn the component server and client, run the experiment, and display the required results. BGML is built on top of the Generic Modeling Environment (GME) [45], which provides a meta-programmable framework for creating domain-specific modeling languages and generative tools. GME is programmed via *meta-models* and *model interpreters*. The

meta-models define modeling languages called paradigms that specify allowed modeling elements, their properties, and their relationships. Model interpreters associated with a paradigm can also be built to traverse the paradigm's modeling elements, performing analysis and generating code.

BGML model elements

To capture QoS evaluation concerns of different component middleware solutions, the BGML provides:

- **Build elements** such as project, workspace, resources and implementation artifact that can be used to represent projects and their dependencies such as DLLs, shared objects. For example, the projects modeled can be mapped to Visual Studio project files (Windows platforms) or onto GNUMake/Make files (*NIX platforms). The build commands can be used to represent compilers for different platforms such as gcc, g++ and ant. Appendix A illustrates the build files generated by BGML.
- **Test elements** such as operations, return-types, latency and throughput that can be used to represent generic operation or a sequence or operation steps and associate functional QoS properties with them. For example, the operation signature (name, input parameters and return-type) can be used to generate platform specific benchmarking code via language mappings (C++/Java) during the interpretation process.
- **Workload elements** such as tasks and task-set that can be used to model and simulate background load present during the experimentation process. These workload elements are then mapped to individual platform specific code in the interpretation process. Appendix A illustrates the generated work-load and benchmark files from BGML.

- **QoS elements** In addition to building blocks required to model the experiment, BGML provides constructs to associate QoS parameters with the experiment modeled. BGML associates QoS parameters with EventTypes on a per-operation basis with IDL interfaces. The metrics that can be gathered via BGML include: **Latency** – For a given operation/event, this metric computes the mean roundtrip time at client.[2]. **Throughput** – For a given operation/event, this metric computes the mean number of invocations completed per-second at the client. **Jitter** – For a given operation/event, this metric computes the variance in the latency measures. Each metric describe above has two attributes: (1) *warmup iterations*, which indicates the number of iterations the operation under test will be invoked before start of actual measurement, and (2) *sample space*, which indicates the number of sample points that will be collected to determine each metric value.

BGML has been integrated with a MDD toolchain called CoSMIC [28] as shown in Figure V.3 and can be downloaded from `www.dre.vanderbilt.edu/cosmic/`. In particular, BGML leverages capabilities provided by other modeling paradigms in CoSMIC including the *Options Configuration Modeling Language* (OCML) [**?**], which is an MDD tool that simplifies the specification and validation of complex DRE middleware and application configurations, and *The Platform Independent Component Modeling Language* [4] tool which supports visual modeling of components, ports, interfaces, and operations.

Integrating BGML With Skoll

This section describes how BGML modeling tools interact with the existing Skoll infrastructure to enhance its DCQA capabilities referencing the steps shown in Figure V.4.

[2]Here a client refers to a CORBA 2.x client or a CCM component playing the role of a client.

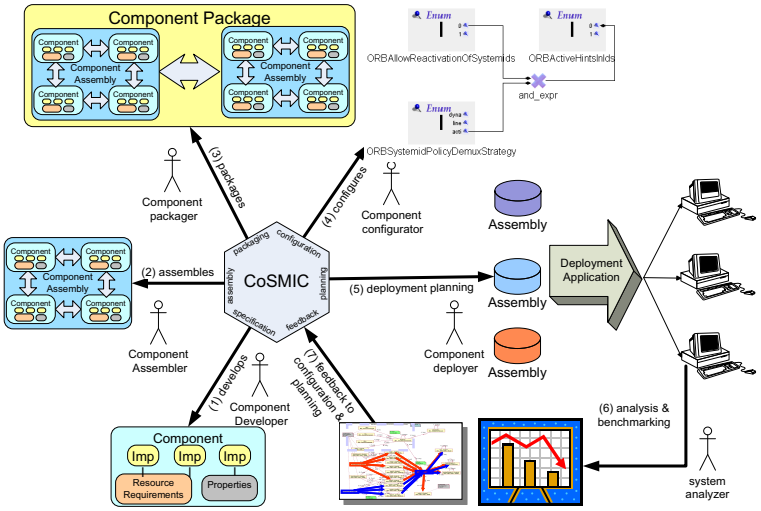

Figure V.3: CoSMIC MDD Tool Chain

1. QA engineers define a test configuration using BGML models. The necessary experimentation details are captured in the models, *e.g.*, the ORB configuration options used, the IDL interface exchanged between the client and the server, and the benchmark metric performed by the experiment.

2. QA engineers then use BGML to interpret the model. The OCML paradigm interpreter parses the modeled ORB configuration options and generates the required configuration files to configure the underlying ORB. The BGML paradigm interpreter then generates the required benchmarking code, *i.e.*, IDL files, the required header and source files, and necessary script files to run the experiment. Steps A, B, and C are integrated with Step 1 of the Skoll process.

3. When users register with the Skoll infrastructure they obtain the Skoll client software and configuration template. This step happens in concert with Step 2, 3, and 4 of the Skoll process.

4. The client executes the experiment and returns the result to the Skoll server, which updates its internal database. When prompted by developers, Skoll displays execution results using an on demand scoreboard. This scoreboard displays graphs and charts

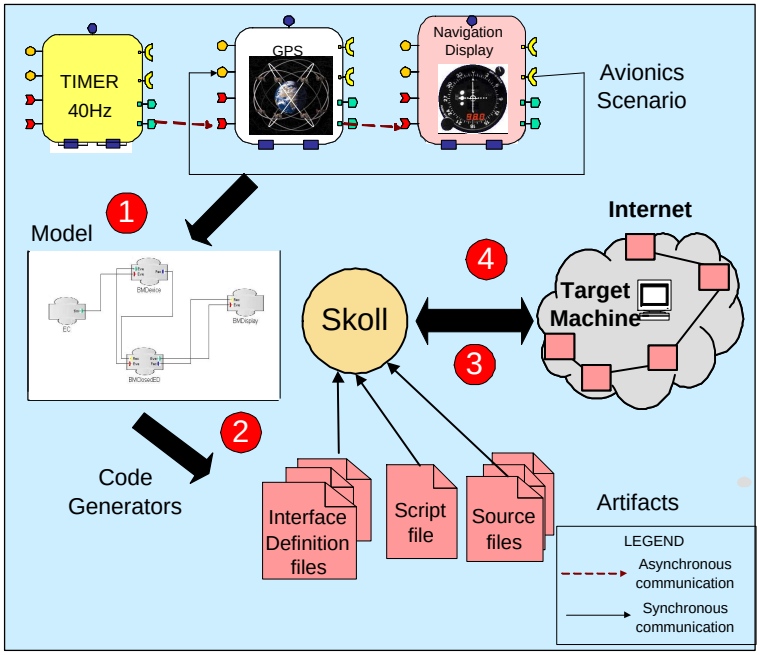

Figure V.4: Model Driven DCQA Approach

for QoS metrics, *e.g.*, performance graphs, latency measures and foot-print metrics. Steps E and F correspond to steps 5, 6, and 7 of the Skoll process.

Applying Model-driven DCQA Process - A Case Study

In this section, BGML and Skoll infrastructure have been applied to execute a formally-designed experiment using a *full-factorial design*, which executes the experimental task (benchmarking in this case) exhaustively across all combinations of the experimental options (a subset of the configuration parameters of the CIAO QoS-enabled component middleware).

The data from these experiments is returned to the Skoll server, where it is organized into a database. The database then becomes a resource for developers of applications and middleware who wish to study the system's performance across its many different configurations. Since the data is gathered through a formally-designed experiment, statistical methods (*e.g.*, analysis of variance, wilcox ran sum tests, and classification tree analysis) are used to analyze the data. To demonstrate the utility

of this approach, two use cases are presented that show how (1) CIAO developers can query the database to improve the performance of the component middleware software and (2) application developers can fine-tune CIAO's configuration parameters to improve the performance of their software.

Overview of Classification Trees

Classification trees use a recursive partitioning approach to build a model that predicts a configuration's class (*e.g.*, passing or failing) in terms of the values of individual option settings. This model is tree-structured. Each node denotes an option, each edge represents an option setting, and each leaf represents a class or set of classes (if there are more than 2 classes).

Classification trees are constructed using data called the *training set*, which consists of configurations, each with the same set of options, but with potentially different option settings together with known class information. Based on the training set, models are built as follows. First, for each option the training set is partitioned based on the option settings. The resulting partition is evaluated based on how well the partition separates configurations of one class from those of another. Commonly, this evaluation is realized as an entropy measure [8].

The option that creates the best partition becomes the root of the tree. To this node an edge for each option setting is added. Finally, for each subset in the partition, the process is repeated. The process stops when no further split is possible (or desirable). Our system uses the Weka implementation of J48 classification tree algorithm with the default confidence factor of 0.25 [103] to obtain the models.

To interpret the model, each path from the root to leaf denotes a set of rules. The interpreter begins with the option at the root of the tree and follows the edge corresponding to one option setting. This process continues until a leaf is encountered.

The class label found at the leaf is interpreted as the predicted class for configurations having same combinations of option settings as those found on the path.

Hypotheses

The use cases presented in this section explore the following hypotheses:

1. The Skoll grid can be used together with BGML to quickly generate benchmark experiments that pinpoint specific QoS performance aspects of interest to developers of middleware and/or applications, *e.g.*, BGML allows QA process engineers to quickly setup QA processes and generate significant portions of the required benchmarking code.
2. Using the output of BGML, the Skoll infrastructure can be used to (1) quickly execute benchmarking experiments on end-user resources across a Skoll grid and (2) capture and organize the resulting data in a database that can be used to improve the QoS of performance-intensive software.
3. Developers and users of performance-intensive software can query the database to gather important information about that software, *e.g.*, obtain a mix of configuration option settings that improve the performance for their specific workload(s).

Experimental Process

The following experimental process was used to evaluate the hypotheses outlined in Section V:

Step 1: Choose a software system that has stringent performance requirements. Identify a relevant configuration space.

Step 2: Select workload application model and build benchmarks using BGML.

Step 3: Deploy Skoll and BGML to run benchmarks on multiple configurations using a full factorial design of the configuration options. Gather performance data.

Step 4: Formulate and demonstrate specific uses of the performance results database from the perspective of both middleware and application developers.

Step 1: Subject Applications

The ACE+TAO+CIAO middleware was used for this study. ACE, TAO, and CIAO run on a wide range of OS platforms, including most versions of UNIX, Windows, and real-time operating systems, such as Sun/Chorus ClassiX, LynxOS, and VxWorks. CIAO adds component support to TAO [87], which is distribution middleware that implements key patterns [88] to meet the demanding QoS requirements of DRE systems.

Step 2: Build Benchmarks

Figure V.5 describes how the ACE+TAO+CIAO QA engineers used the BGML tool to generate the screening experiments to quantify the behavior of latency and throughput.

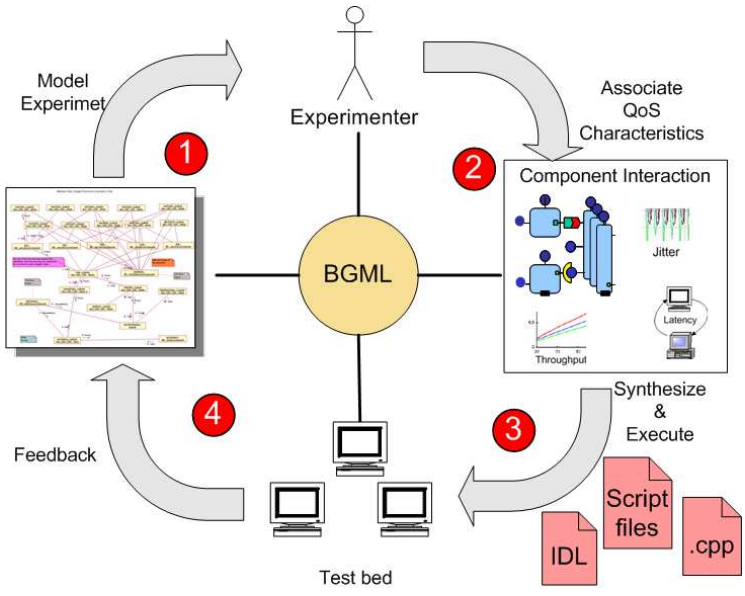

Figure V.5: BGML Use Case Scenario

As shown in this figure, the following steps were performed:

1. QA engineers used the BGML modeling paradigm to compose the experiment. In particular, QA engineers use the domain-specific building blocks in BGML to compose experiments.
2. In the experiment modeled, QA engineers associated the QoS characteristic (in this case roundtrip latency and throughput) that will be captured in the experiment. Figure V.6 depicts how this is done in BGML.
3. Using the experiment modeled by QA engineers, BGML interpreters generated the benchmarking code required to set-up, run, and tear-down the experiment. The generated files include component implementation files (.h and .cpp), IDL files (.idl), component IDL files (.cidl), and benchmarking code (.cpp) files.
4. The generated file was then executed using the Skoll DCQA process and QoS characteristics were measured. The execution was done in Step 4 described in Section V.

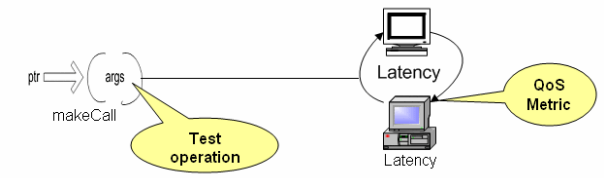

Figure V.6: Associating QoS Metrics in BGML

Step 3: Execute the DCQA process

For this version of ACE+TAO+CIAO, 14 run-time options were identified that could affect latency and throughput. As shown in Table V.1, each option is binary, so the entire configuration space is $2^{14} = 16,384$. The benchmark experiments were

executed on each of the 16,384 configurations. This is called a full-factorial experimental design. Clearly such designs will not scale up to arbitrary numbers of factors. In the current example, however, the design is manageable.

Table V.1: The Configuration Space: Run-time Options and their Settings

Option Index	**Option Name**	**Option Settings**
opt1	ORBReactorThreadQueue	{FIFO, LIFO}
opt2	ORBClientConnectionHandler	{RW, MT}
opt3	ORBReactorMaskSignals	{0, 1}
opt4	ORBConnectionPurgingStrategy	{LRU, LFU}
opt5	ORBConnectionCachePurgePercentage	{10, 40}
opt6	ORBConnectionCacheLock	{thread, null}
opt7	ORBCorbaObjectLock	{thread, null}
opt8	ORBObjectKeyTableLock	{thread, null}
opt9	ORBInputCDRAllocator	{thread, null}
opt10	ORBConcurrency	{reactive, thread-per-connection}
opt11	ORBActiveObjectMapSize	{32, 128}
opt12	ORBUseridPolicyDemuxStrategy	{linear, dynamic}
opt13	ORBSystemidPolicyDemuxStrategy	{linear, dynamic}
opt14	ORBUniqueidPolicyReverseDemuxStrategy	{linear, dynamic}

For a given configuration, the BGML modeling paradigm was used to model the configuration visually and generate the scaffolding code to run the benchmarking code. The experiment was run three times and for each run the client sent 300,000 requests to the server. In total, ∼50,000 benchmarking experiments were run. For each run, the latency values for each request and total throughput (events/second) was measured.

The BGML modeling tool helps improve the productivity of QA engineers by allowing them to compose the experiment *visually* rather than wrestling with low-level source code. This tool thus resolves tedious and error-prone accidental complexities associated with writing correct code by auto-generating them from higher level models. Table V.2 summarizes the BGML code generation metrics for a particular configuration.

This table shows how BGML automatically generates 8 of 10 required files that

Table V.2: Generated Code Summary for BGML

Files	Number	Lines of Code	Generated (%)
IDL	3	81	100
Source (.cpp)	2	310	100
Header (.h)	1	108	100
Script (.pl)	1	115	100

account for 88% of the code required for the experiment. Only the XML descriptor files must be written manually. This improvement is accrued for each of the 16,348 configurations, cumulatively representing an order of magnitude improvement in productivity for performing benchmarking QA tasks.

Step 4: Example Use Cases

Below two use cases that leverage the data collected by the Skoll DCQA process are presented. The first scenario involves application developers who need information to help configuring CIAO for their use. The second involves CIAO middleware developers who want to prioritize certain development tasks.

Use case #1: Application developer configuration. In this scenario, a developer of a performance-intensive software application is using CIAO. This application is expected to have a fairly smooth traffic stream and needs high overall throughput and low latency for individual messages. This developer has decided on several of the option settings needed for his/her application, but is unsure how to set the remaining options and what effect those specific settings will have on application performance. To help answer this question, the application developer goes to the ACE+TAO+CIAO Skoll web page and identifies the general workload expected by the application, the platform, OS, and ACE+TAO+CIAO versions used. Next, the developer arrives at the web page shown in Figure V.7.

On this page the application developer inputs those option settings (s)he expects to use and left unspecified (denoted "*") those for which (s)he needs guidance. The

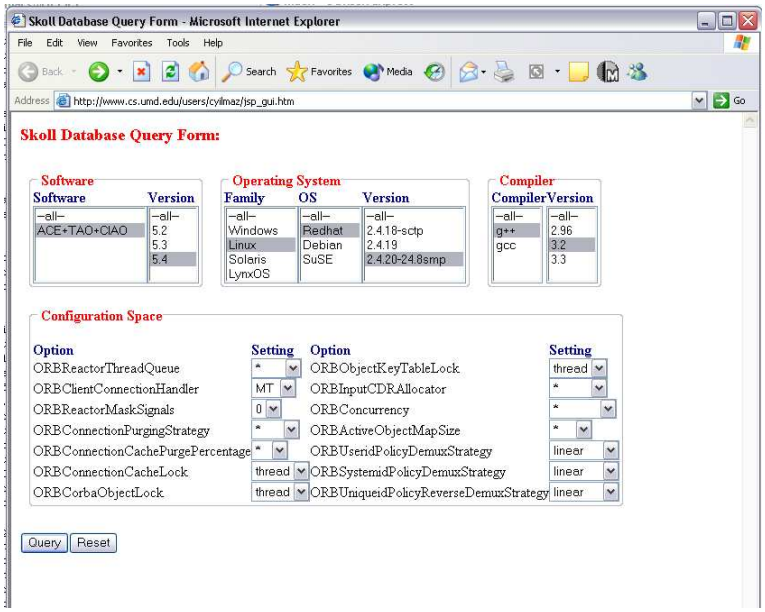

Figure V.7: Accessing Performance Database

developer also indicates the performance metrics (s)he wishes to analyze and then submits the page.

Submitting the page causes several things to happen. First, the data corresponding to the known option settings is located in the Skoll databases. Next, the system graphs the historical performance distributions of both the entire configuration space and the subset specified by the application developer (*i.e.*, the subset of the configuration space consistent with the developer's partially-specified options). These graphs are shown in Figure V.8 and Figure V.9.

Last, the system presents a statistical analysis of the options that significantly affect the performance measures, as depicted in Figure V.9. Together, these views present the application developer with several pieces of information. First, it shows how the expected configuration has performed historically on a specific set of benchmarks. Next, it compares this configuration's performance with the performance of other possible configurations. It also indicates which of the options have a significant effect on performance and thus should be considered carefully when selecting the final configuration.

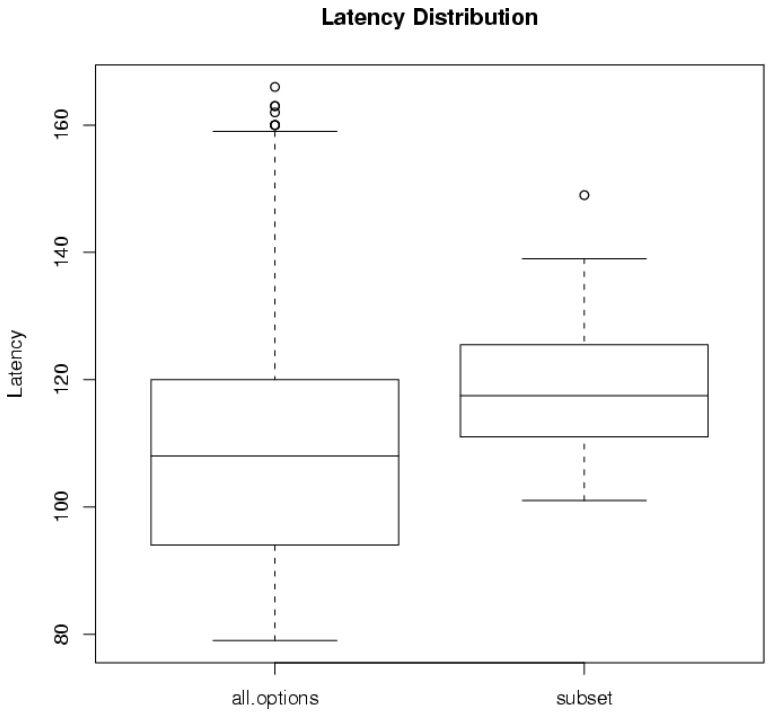

Figure V.8: 1^{st} Iteration

Continuing the use case example, the application developer sees that option *opt10* (`ORBConcurrency`) has not been set and that it has a significant effect on performance. To better understand the effect of this option, the developer consults the main effects graph shown in Figure V.9). This plot shows that setting `ORBConcurrency` to *thread-per-connection* (where the ORB dedicates one thread to each incoming connection) should lead to better performance than setting it to *reactive* (where the ORB uses a single thread to detect, demultiplex and service multiple client connections). The application developer therefore sets the option and reruns the earlier analysis. The new analysis shows that, based on historical data, the new setting does indeed improve performance, as shown in Figure V.10.

However, the accompanying main effects graph shown in Figure V.11 shows that the remaining unset options are unlikely to have a substantial effect on performance. At this point, the application developer has several choices, *e.g.*, (s)he can stop here

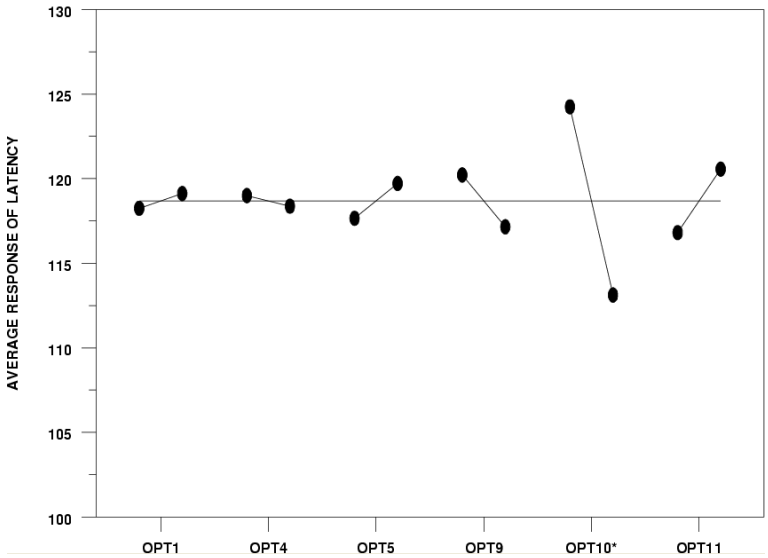

Figure V.9: 1^{st} Iteration: Main Effects Graph (Statistically Significant Options are Denoted by an *)

and set the remaining options to their default settings or (s)he can revisit the original settings. In this case, the developer reexamines the original settings and their main effects (See Figure V.12) and determines that changing the setting of *opt2* (`ORBClientConnectionHandler`) might greatly improve performance.

Using this setting will require making some changes to the actual application, so the application developer reruns the analysis to get an idea of the potential benefits of changing the option setting. The resulting data is shown in Figure V.13. The results in this figure show that the performance improvement from setting this option would be substantial. The developer would now have to decide whether the benefits justify the costs of changing the application.

Use case #2: Middleware developer task prioritization. In this scenario, a developer of CIAO middleware itself wants to do an exploratory analysis of the system's performance across its configuration space. This developer is looking for areas that are in the greatest need of improvement. To do this (s)he accesses the ACE+TAO+CIAO and Skoll web page and performs several tasks.

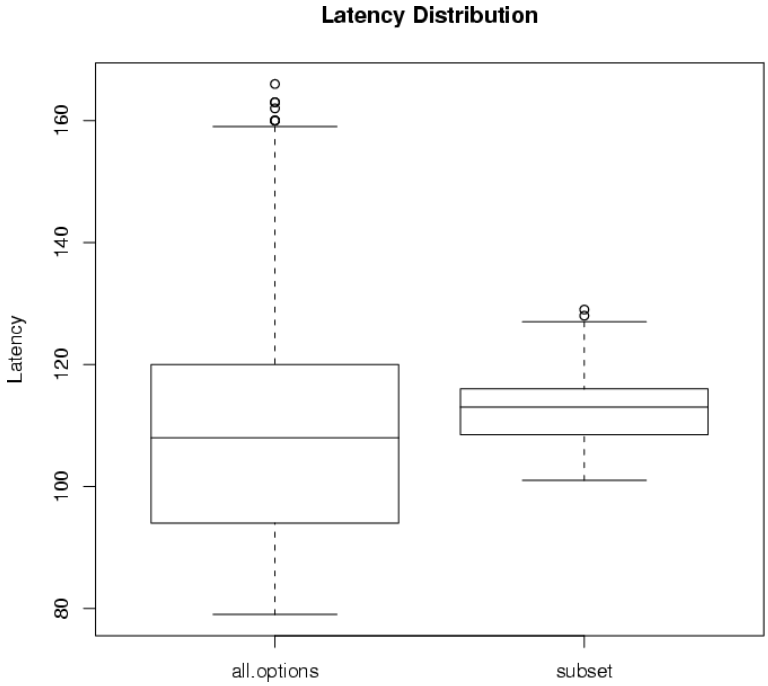

Figure V.10: 2^{nd} Iteration

First, (s)he examines the overall performance distribution of one or more performance metric. In this case, the middleware developer examines measurements of system latency, noting that the tails of the distribution are quite long (the latency plots are the same as those found in the "all.options" subplots of Figure V.8). The developers wants to better understand which specific configurations are the poor performers.[3]

Our DCQA process casts this question as a classification problem. The middleware developer therefore recodes the performance data into two categories: those in the worse-performing 10% and the rest. From here out, (s)he considers poor performing configurations as those in the bottom 10%. Next, (s)he uses classification tree analysis [66] to model the specific combinations of options that lead to degraded performance. Sidebar V describes how classification trees work.

For the current use case example, the middleware developer uses a classification

[3]For latency the worst performers are found in the upper tail, whereas for throughput it is the opposite.

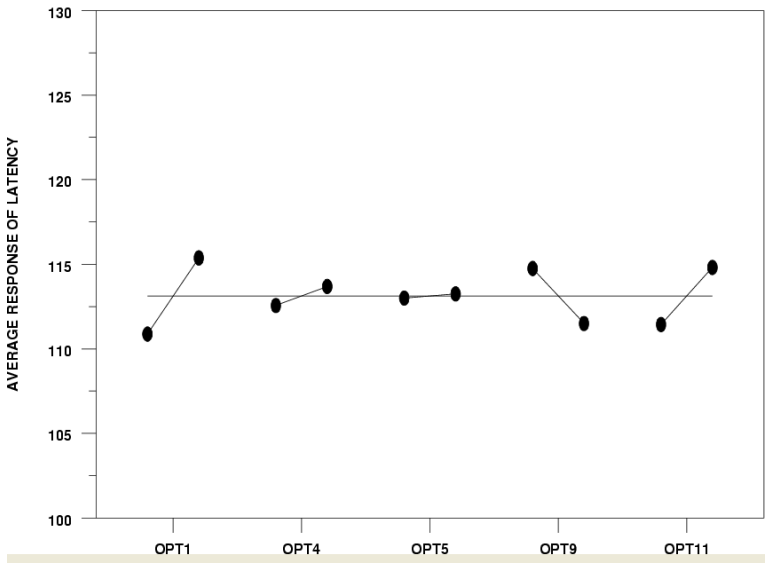

Figure V.11: 2^{nd} Iteration: Main effects graph (Statistically Significant Options are Denoted by an *)

tree to extract performance-degrading option patterns, *i.e.*, (s)he extracts the options and option settings from the tree that characterize poorly performing configurations. Figure V.14 shows one tree obtained from the CIAO data (for space reasons the tree shown in the Figure gives only a coarse picture of the information actually contained in the tree).

By examining the tree, the middleware developer notes that a large majority of the poorly performing configurations have `ORBClientConnectionHandler` set to *MT* and `ORBConcurrency` set to *reactive*. The first option indicates that the CORBA ORB uses separate threads to service each incoming connections. The second option indicates that the ORB's reactor [88] (the framework that detects and accepts connections and dispatches event to the corresponding event handlers when events arrive) are executed by a pool of threads.

The information gleaned by the classification tree is then used to guide exploratory data analysis. To help middleware developers organize and visualize the large amount of data, the Treemaps data visualizer (`www.cs.umd.edu/hcil/treemap`) is employed,

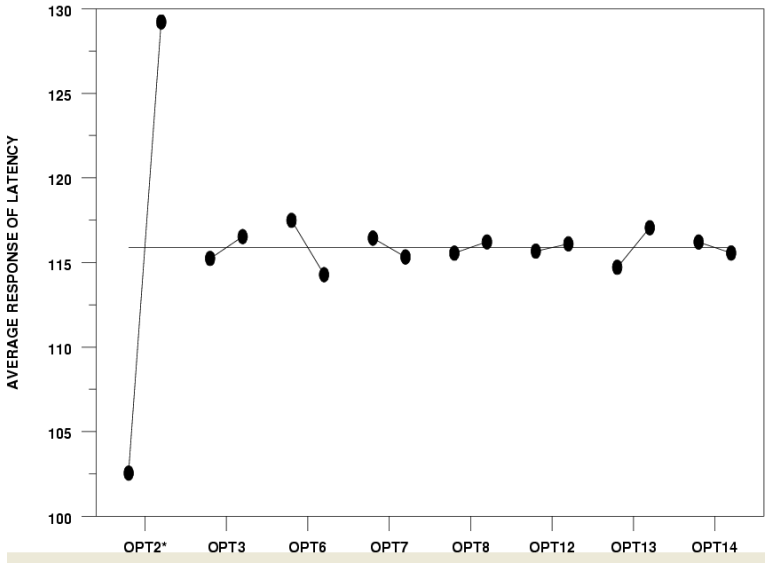

Figure V.12: 3^{rd} Iteration: Main Effects Graph (Statistically Significant Options are Denoted by an *)

which allows developers to explore multidimensional data. The performance data described in the previous paragraph is shown in Figure V.15.

This figure shows poorly performing configurations as dark tiles and the acceptably performing configurations as lighter tiles. The layout first divides the data into two halves: the left for configurations with `ORBClientConnectionHandler` set to *RW* and the right for those set to *MT*. Each half is further subdivided, with the upper half for configurations with `ORCConcurrency` set to *thread-per-connection* and the lower half for those set to *reactive*. The data can be further subdivided to arbitrary levels, depending on how many options the middleware developer wishes to explore. The treemap shown in Figure V.15 depicts how almost all the poor performers are in the bottom right quadrant, which suggests that the options discovered by the classification tree are reasonably good descriptors of the poorly performing configurations.

The middleware developer continues to explore the data, checking whether the addition of other options would further isolate the poor performers, thereby providing more information about the options that negatively influence performance. After

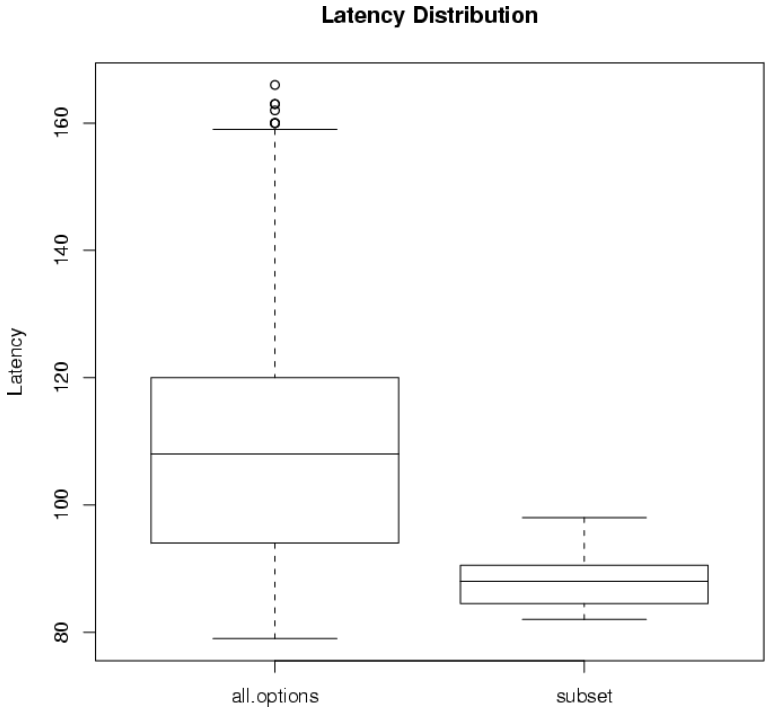

Figure V.13: 3^{rd} Iteration: Step 3

some exploration, the middleware developer find no other influential options. Next, (s)he examines the poor performing configurations that are not part of the main group, *i.e.*, those with `ORBCurrency` set to *thread-per-connection* rather than *reactive*. The middleware developer determines that nearly all of the latency values for these configurations are quite close to the 10% cutoff. In fact, lowering the arbitrary cutoff to around 8% leads to the situation in which nearly every poor performer has `ORBConnectionClientHandler` set to *MT* and `ORBConcurency` set to *reactive*. Based on this information, the middleware developer can conduct further studies to determine whether a redesign might improve performance.

Summary

Reusable software for performance-intensive systems increasingly has a multitude of configuration options and runs on a wide variety of hardware, compiler, network,

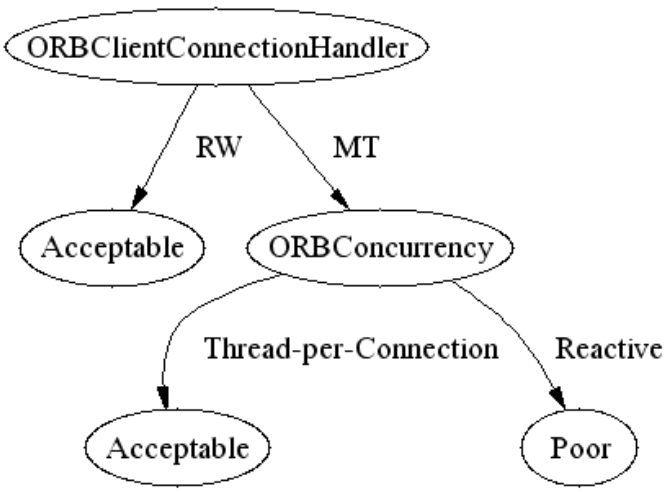

Figure V.14: Classification Tree Modeling Poorly Performing Configurations

OS, and middleware platforms. This variability has yielded an explosion in the configuration and platform space on which the quality assurance (QA) of the software must be evaluated. The model-driven DCQA techniques play an important role in ensuring the correctness and quality of service (QoS) of performance-intensive software.

DCQA approaches helps to ameliorate the variability in reusable software contexts by providing

- Domain-specific modeling languages that encapsulate the variability in software configuration options and interaction scenarios within GME modeling paradigms.
- An Intelligent Steering Agent (ISA) to map configuration options to clients that test the configuration and adaptation strategies to learn from the results obtained from clients and
- Model-based interpreters that generate benchmarking code and provide a framework to automate benchmark tests and facilitate the seamless integration of new tests.

The work on Skoll addresses two key dimensions of applying distributed continuous quality assurance (DCQA) processes to reusable performance-intensive software.

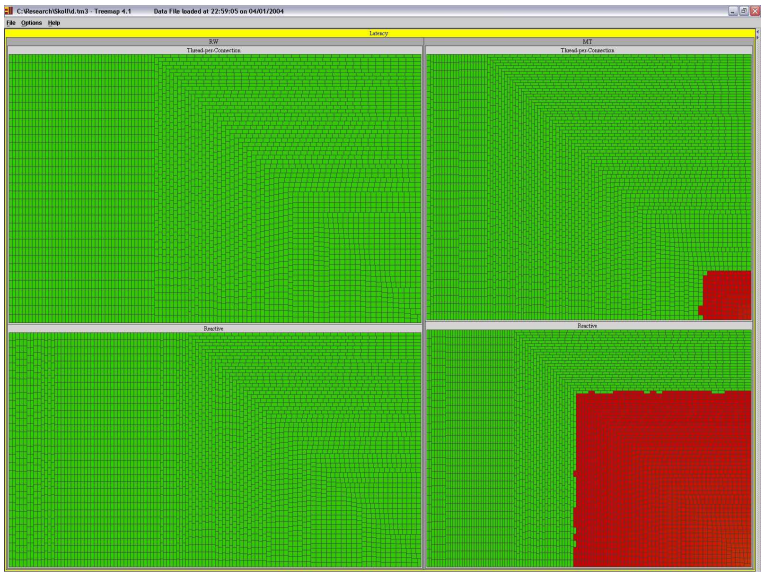

Figure V.15: Treemap Visualization

The skoll infrastructure address software functional correctness issues, *e.g.*, ensuring software compiles and runs on various hardware, OS, and compiler platforms. The MDD tools address software QoS issues, *e.g.*, modeling and benchmarking interaction scenarios on various platforms by mixing and matching configuration options. These model-based QA techniques enhance Skoll by allowing developers to model configuration/interaction aspects and associate metrics to benchmark the interaction. These techniques also minimize the cost of testing and profiling new configurations by moving the complexity of writing error-prone code from QA engineers into model interpreters, thereby increasing productivity and quality. Model-based tools such as BGML simplify the work of QA engineers by allowing them to focus on domain specific details rather than write source code.

Our experimental results showed how the modeling tools improve productivity by resolving the accidental complexity involved in writing error-prone source code for each benchmarking configuration. Section V showed that by using MDD approach,

$\sim$90% of the code required to test and profile each combination of options can be generated, thereby significantly reducing the effort required by QA engineers to empirically evaluate impact of software variability on numerous QOS parameters. Section V showed how the results collected using Skoll can be used to populate a data repository that can be used by both application and middleware developers. The two use case presented in this feasibility study showed how this approach provides feedback to (1) application developers, *e.g.*, to tune configurations to maximize end-to-end QoS and (2) middleware developers, *e.g.*, to more readily identify configurations that should be optimized further.

CHAPTER VI

CONCLUDING REMARKS AND FUTURE RESEARCH DIRECTIONS

This dissertation focused on techniques for enhancing middleware QoS for software product-line architectures. Product-line architectures (PLAs) enable organizations to reconfigure their software quickly to respond to new missions and new business opportunities [12]. PLAs are particularly applicable for large-scale distributed real-time and embedded (DRE) systems since reusable software families can be built for a domain (such as avionics, vetronics, or ship-board computing) where applications share many functional and architectural properties and then customized to meet the specific needs of product variants. Standards-based middleware is a key infrastructure technology for PLAs since it provides many reusable policies and mechanisms to simplify the development, customization, and deployment of product variants. The stringent demands of DRE systems, however, require conservation of resources, while simultaneously providing the desired QoS. It is therefore essential to customize standards-based middleware implementations by eliminating extraneous functionality and extensibility not needed for particular product variants. The OPTEML approach resolved three key dimensions of challenges pertaining to customizing general-purpose standards based middleware for PLAs via:

- **Fine-grain componetization techniques**, that transparently allow middleware implementations to be full featured yet minimal in footprint,
- **Specialization techniques**, that remove extraneous time/space overhead within middleware based on functional and QoS requirements of PLAs, and
- **Validation techniques**, that ensure the correctness of componentized and specialized middleware implementations across different hardware, OS and compiler platforms.

The remainder of this chapter discusses how OPTEML has been integrated and validated using representative middleware solutions, summarizes the research contributions and lessons learned and outlines future research directions.

Research Integration & Validation

OPTEML has been integrated and validated on several representative applications using middleware developed with Real-time Java and C++. The fine-grain middleware componentization techniques have been validated using the ZEN (`www.zen.uci.edu`) ORB which is an open-source Real-time CORBA ORB implemented using the RTSJ and is being used as the middleware implementation in the DARPA PCES program. The specialization techniques have been implemented and validated using the open-source TAO (`www.cs.wustl.edu/~schmidt/TAO.html`) implementation of Real-time CORBA written in C++. TAO 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`). To automate the specialization a new open-source Feature Oriented Customizer Tool (FOCUS) has been developed. FOCUS has been applied to ACE (`www.cs.wustl.edu/~schmidt/ACE.html`) and TAO middleware implementations. ACE [84, 85] implements core concurrency and distribution patterns [88] for communication software and is the host infrastructure middleware on which TAO is implemented. FOCUS is available with the ACE+TAO distribution and can be downloaded from `http://deuce.doc.wustl.edu/Download.html`.

The BGML Domain Specific Modeling Language (DSML) has been integrated with the CoSMIC MDD tool chain `www.dre.vanderbilt.edu/cosmic`. CoSMIC is a collection of DSMLs and generative tools that support the development, configuration, deployment, and validation of component-based DRE systems. BGML has been applied to CIAO [102] which is a QoS-enabled implementation of CCM to help

simplify the development of performance-intensive software applications by enabling developers to declaratively provision QoS policies end-to-end when assembling a DRE system. Finally, BGML works in conjunction with the Skoll `www.cs.umd.edu/skoll` DCQA tool which is begin used to validate the specializations techniques presented in this dissertation on a range of hardware, OS and compiler platforms.

Lessons Learned & Research Contributions

The following are the lessons learned from OPTEML. The middleware componentization techniques illustrated that middleware subsetting should be planned early in the design cycle. Reducing middleware footprint after the middleware has become mature increases the complexity involved in subsetting middleware. Such componentization should also be transparent to the PLA application developers and should not modify the standard CORBA interfaces. In ZEN for example, the POA componentization case study showed how use of virtual component pattern allows the POA to be full featured yet minimal in footprint.

The specialization techniques showed that general-purpose standards based middleware has extraneous functionality that is undesirable for PLA based DRE applications. The specialization techniques, however, should not affect portability, standard middleware APIs, or application software implementations, and preserve interoperability wherever possible. Solution approaches that do not honor the above conditions obviate the benefits accrued from using standards based middleware. In addition, accidental complexity from manually applying this specialization to mature middleware implementations renders the specializations tedious and error prone to implement. Specialization automation is essential for making the specializations viable to mature, feature rich and large middleware code bases like TAO.

With middleware evolution, it is necessary to validate and quantifying impact of specializations on performance across different hardware, OS and compiler platforms.

The Skoll and BGML approaches, described how model-driven and distributed continuous QA approaches can be combined to provide a QA framework validating the specializations approach. In summary this dissertation has made the following contributions to the research on optimizing, specializing and validating middleware for PLAs.

1. It shows how *context-specific specialization techniques* (such as code refactoring, and code weaving) can be used to customize the widely used TAO 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. It describes the design of a domain-specific language, tools, and a process for *automating the specialization techniques* discussed in the dissertation.
2. It describes novel model-driven distributed continuous quality assurance approaches for validating the componentized and specialized middleware across a range of hardware, OS and compiler platforms.
3. It presents qualitative and quantitative results that demonstrate the benefits of applying the different dimensions of research discussed in the dissertation.

Figure VI.1 categorizes the research contributions hierarchically. At the base are novel optimization strategies, patterns, and idioms that help componentize middleware implementation. At the next level are middleware specialization techniques that transparently specialize middleware to remove unnecessary generality for PLAs. These techniques improve middleware QoS including end-to-end latency, throughput and jitter. Finally, the DCQA approaches discussed in this paper help in validating componentized and specialized middleware on varying hardware, OS and compiler platforms.

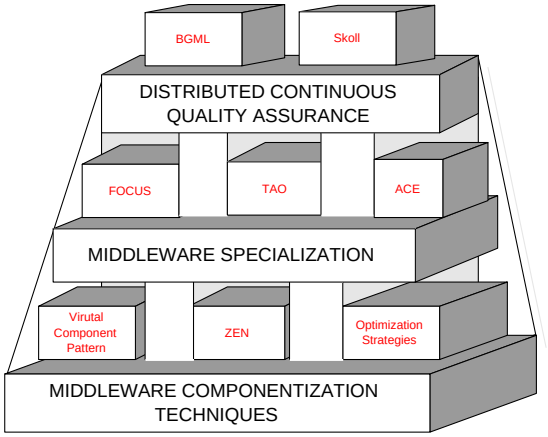

Figure VI.1: Research Contributions

Future Research Directions

Recent progress in *QoS-enabled component middleware* [102] has shown great promise for DRE systems by enabling reusable services to be composed, configured, and installed to create applications rapidly and robustly. *Component middleware* [96] is a class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly. Conventional component middleware platforms, such as J2EE and .NET, is not well-suited for these types of DRE systems since they do not provide real-time quality of service (QoS) support.

QoS-enabled component middleware, such as CIAO [101], Qedo [73], and PRiSm [91], have been developed to address these limitations by combining the flexibility of component middleware with the predictability of Real-time CORBA. The OMG's Deployment and Configuration specification [63] enhances component middleware in order to (1) deploy component assemblies into the appropriate DRE system target nodes, (2) activate and deactivate component assemblies automatically, (3) initialize and configure component server resources to enforce end-to-end QoS requirements of component assemblies, and (4) simplify the configuration, deployment, and management of common services used by applications and middleware. The *Deployment and Configuration Engine* (DAnCE), is an open-source (`www.dre.vanderbilt.edu/CIAO`)

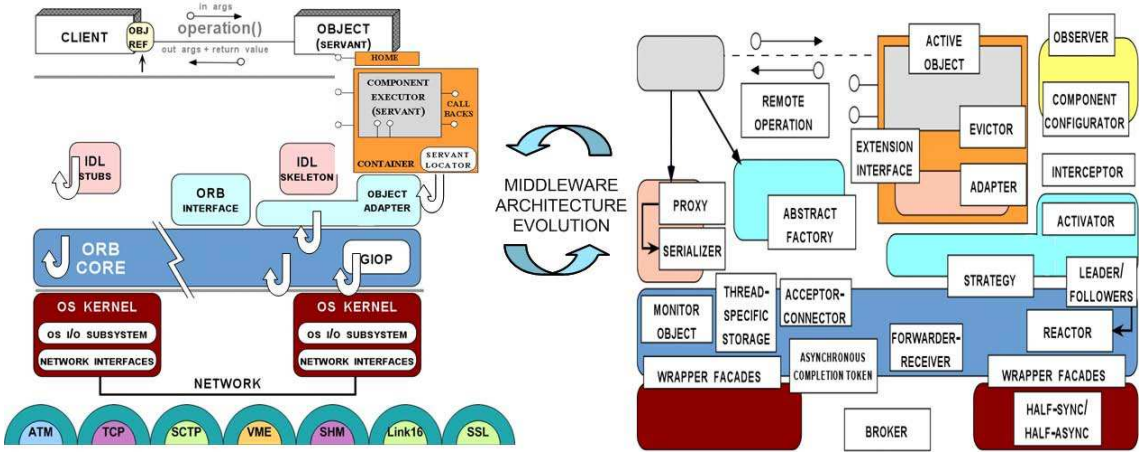

Figure VI.2: Middleware Evolution & Variability

QoS-enabled middleware framework compliant with the OMG Deployment and Configuration specification [63]. The remainder of this section describes how OPTEML approach can be synergistically combined with component middleware in general and DAnCE in particular.

Specializing Middleware Frameworks

Middleware comprise different frameworks that are amenable to specialization via our FOCUS approach. Figure VI.2 illustrates the variability points within middleware as configurable hooks. The figure also illustrates how patterns and pattern languages have been applied to accommodate this variability. FOCUS approach presented in this paper specialized the Reactor and protocol frameworks in TAO. Other opportunities for specialization exists within middleware including, locking, concurrency, demultiplexing and dispatching frameworks in TAO and middleware. Future research will therefore need to investigate how specialization approaches can be applied to other middleware frameworks and services to enhance application QoS.

Specializing Component Middleware Implementations

Today's middleware only provides the assembly and deployment of application-level components [28] hosted by DRE middleware [102]. There exists a need for research to significantly advance DRE software systems by developing and validating novel ideas for bootstrapping the QoS-enabled component middleware itself. This research illustrated how specialization approaches can be applied to DOC middleware. Similar opportunities exists in the component middleware as well. For example, in CCM, a *container* provides the execution environment need to execute components in generic component servers as shown in Figure VI.3.

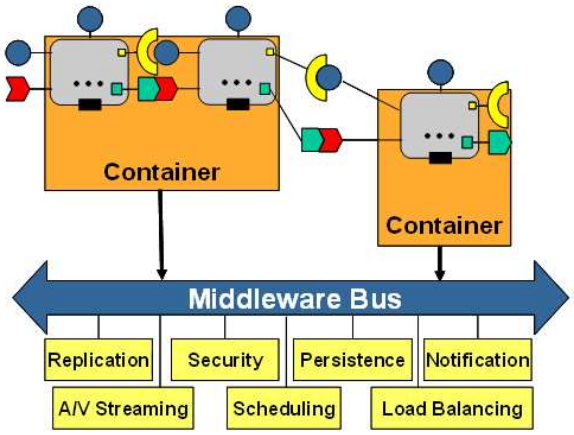

Figure VI.3: CCM Container

To suite different applications, containers provide configurable QoS hooks via policies. Containers also allow components to access middleware services such as event-, persistence- and security service. Future research, will therefore need to investigate mechanisms, similar to policy driven specialization approaches discussed in this dissertation to provide a full featured container yet incurring minimal footprint.

Managing Specialized Middleware and Component Implementation

DAnCE's `RepositoryManager` provides efficient mechanisms where applications can (1) store component implementations at any time during the system lifecycle and (2) retrieve different versions of implementations as components are (re)deployed on various types of nodes. As shown in Figure VI.4, the `RepositoryManager` can also act as an HTTP client and download component implementations specified as *URLs* in a deployment plan. It caches these implementations in the local host where the `RepositoryManager` runs so they can be retrieved by processes on individual hosts *i.e.*, `NodeApplicationManagers`.

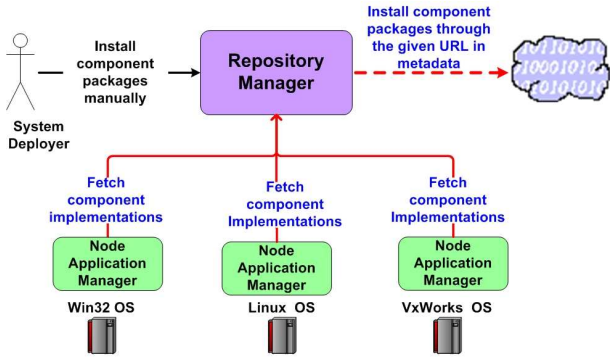

Figure VI.4: DAnCE Repository Manager

The FOCUS toolkit presented in the dissertation, allows creation of specialized middleware implementations based on PLA system invariance. DAnCE's `Repository Manager` can be used to store different specialized middleware implementations. During component deployment, the `Repository Manager` can be used to retrieve specialized middleware and component libraries for deployment. Future research will therefore need to develop mechanisms on how middleware specializations can be integrated with Deployment and Configuration specification.

Model-Driven Specialization Approaches

Customizing middleware for different operating contexts requires a systematic approach to designing and implementing different context-specific specializations. Model-driven Development (MDD) offers the right choice to realize a systematic and scientific approach to resolving the PLA middleware challenges described in this dissertation. The FOCUS approach detailed a process for executing the specializations. Combining the FOCUS approach with MDD provides a two step process in which (1) identification of the specialization points and transformations are done via Model-driven approaches and (2) (2) automating the delivery of the specializations is done by FOCUS like approach.

Future research will therefore need to focus on:

- **Defining and capturing PLA invariants** – which will require the modeling tool to provide capabilities to capture the PLA-specific feature invariants. Modeling tools will need to map these invariants to the features in middleware that will be required to satisfy these invariants.
- **Represent middleware models and their configurability** – which will require the modeling tool to provide capabilities to represent middleware as building blocks that can be configured and customized according to the PLA and product-specific requirements. This capability is driven by the state of art in middleware, which comprises a composition and configuration of patterns-based building blocks.
- **Capturing product-specific functional and QoS variability** – which will require the ability to specify product-specific variability incurred due to functional and QoS requirements. This will also govern the variability in the assembly, configuration and deployment of the product variant and the associated middleware infrastructure.

Figure VI.5 illustrates a futuristic view of a specialization process.

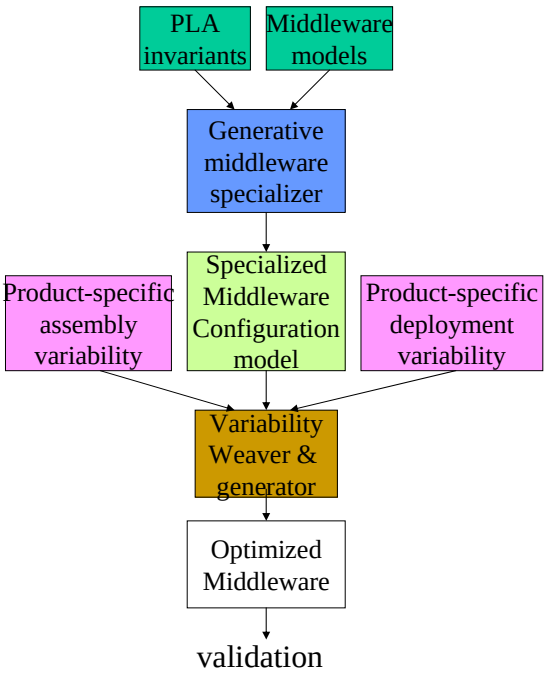

Figure VI.5: Model-driven Middleware Specialization Approach

In this approach, middleware models and PLA invariance specification is fed to a middleware specializer that generates a middleware configuration suitable for the PLA. From this base configuration model, the variability specifications, *e.g.*, the assembly and deployment aspects are woven by a variability weaver generating an optimized middleware implementation. QA Techniques similar to the ones discussed in this dissertation can be applied to validate the synthesized optimized middleware.

APPENDIX A

BGML GENERATED CODE

This section shows the generated code from the BGML model interpreters. Each code snippet, corresponds to a separate capability provided by BGML as described in Section V. The generated code described is specific to the CIAO QoS enabled middleware. Other model interpreters can be associated with these aspects to generate code for other QoS enabled middleware implementations.

Build File code snippet

In this section, shows the format of a build file generate by the Component build interpreter. In order to focus the discussion, only a part of the generated build file is shown.

```
1: project(BMDisplay_stub): ciao_client {
2:  after += BasicSP_stub
3:  sharedname = BMDisplay_stub
4:  idlflags +=
    -Wb,stub_export_macro=BMDISPLAY_STUB_Export
    -Wb,stub_export_include=BMDisplay_stub_export.h
    -Wb,skel_export_macro=BMDISPLAY_SVNT_Export
    -Wb,skel_export_include=BMDisplay_svnt_export.h
5:  dynamicflags = BMDISPLAY_STUB_BUILD_DLL
6:
7:  IDL_Files {
8:    BMDisplay.idl
9:  }
10:
```

```
11: Source_Files {
12:    BMDisplayC.cpp
13: } }
```

The section next explains what the individual lines refer to using the terminology corresponding to the Visual Studio build file. Line 1 shows a project definition for the `BMDisplay_stub` component which inherits from a generic client component (base). This corresponds to the creation of a project in a Visual Studio environment. The second line describes a dependency with another project in the same workspace while `sharedname` describes the name of the DLL created. Line 4 illustrates flags passed to the TAO IDL compiler while Line 7 specifies the name of the idl file. Finally lines 11-13 describe the name of the source implementation file.

Component IDL file code snippet

This section shows the format of the Component idl file generated by the Component IDL interpreter. This IDL file is fed to an idl compiler to generate "glue-code" needed by the ORB infrastructure.

```
1:  #ifndef BMDISPLAY_CIDL
2:  #define BMDISPLAY_CIDL

3:  #include "BMDisplay.idl"

4:  composition session BMDisplay_Impl
5:  {
6:    home executor BMDisplayHome_Exec
7:    {
8:      implements BasicSP::BMDisplayHome;
9:      manages BMDisplay_Exec;
10:   };
```

```
11: };
```

```
12: #endif /* BMDISPLAY_CIDL */
```

Lines 1-3 illustrated the header information. The included IDL file is generated from the models directly by the IDL compiler. Line 5 defines the type of the Component implementation as a "session" component (the other being an entity component). Line 6 defines the a *home* that manages component while Lines 8-9 defines a composition *i.e.*, binds the home to the type of the component.

Benchmark code snippet

This section illustrates the Benchmarking code generated by the BGML model interpreters to benchmark an operation with the following signature: `get_data(longdata)`. The interpreters generate the header and source file that follow the following convention: (1) All files names have the prefix Benchmark and (2) The operation name is used to complete the file name. This convention ensures that the generated file names are always unique. Further, the generated files use templates, to provide a generic benchmarking framework. Below the structure of the header and source files are illustrated.

```
1: template <typename T>
2: class Benchmark_Get_Data : BGML_Task_Base
3: {
4:  public:
5:   Benchmark_Get_Data  (T remote_ref,
                          ::CORBA::Long arg1);
6   ~Benchmark_Get_Data ();

7:  int svc (void);
```

```
8: protected:
9:  T remote_ref_;
10:  ::CORBA::Long arg1_;

11:};
12: #include "Benchmark_Get_Data.cpp"

13: #endif // BENCHMARK_GET_DATA_H
```

Line 2 shows the template `Benchmark_Get_Data` class that uses a generic `BGML_Task_Base` class which provides the capability to associate the class with a thread of a given priority. Using the constructor of the class, the remote reference and the argument are automatically generated (see Line 9). The type of the argument corresponds to the argument of the `get_data` operation. The remote reference is a reference to the target component on which the operation has to be invoked. The `svc` method defined in Line 7 is where the actual benchmarking is done as shown below.

```
template <typename T>
void
Benchmark_Get_Data<T>::svc (void)
{
  // Warmup iterations before benchmarking
  for (int warm_up = 0; warm_up < 100; warm_up++)
    (void) this->remote_ref_->get_data (arg1);

  // Generate the Background workload
  Get_Data_Workload load0 (this->remote_ref_,
                           arg1,
                           0);
```

```
Get_Data_Workload load1 (this->remote_ref_,
                         arg1,
                         0);
// Activate the Background tasks
if (task0.activate (THR_NEW_LWP |
                    THR_JOINABLE, 1, 1) == -1)
 ACE_ERROR ((LM_ERROR, "Error activating task0 \n"));
if (task1.activate (THR_NEW_LWP |
                    THR_JOINABLE, 1, 1) == -1)
 ACE_ERROR ((LM_ERROR, "Error activating task1 \n"));

ACE_Sample_History history (5000);
ACE_hrtime_t test_start = ACE_OS::gethrtime ();
for (i = 0; i < 5000; i++)
{
  ACE_hrtime_t start = ACE_OS::gethrtime ();
  (void) this->remote_ref_->get_data (arg1);
  ACE_CHECK;
  ACE_hrtime_t now = ACE_OS::gethrtime ();

  history.sample (now - start);
}

ACE_hrtime_t test_end = ACE_OS::gethrtime ();
ACE_DEBUG ((LM_DEBUG, "test finished"));
ACE_UINT32 gsf =
  ACE_High_Res_Timer::global_scale_factor ();
ACE_DEBUG ((LM_DEBUG, "done"));
ACE_Basic_Stats stats;
```

```
  history.collect_basic_stats (stats);
  stats.dump_results ("Total", gsf);
  ACE_Throughput_Stats::dump_throughput ("Total",
                                         gsf,
 test_end - test_start,
 stats.samples_count ());
}
```

The actual benchmarking proceeds in three phases (1) given number of warmup iterations are done before the actual benchmarking. For example, the above code snippet uses 100 iterations to warmup the system. (2) A corresponding background load is generated. For example, in the code above four tasks with priority 0 are used to generate background load. (3) Benchmarking target operation and dumping out the corresponding statistics like throughput.

APPENDIX B

LIST OF PUBLICATIONS

Research on OPTEML has lead to the following referred journal, conference and workshop publications.

Referred Journal Publications

1. Arvind S. Krishna, Cemal Yilmaz, Adam Porter, Atif Memon, Douglas C. Schmidt, and Aniruddha Gokhale, Distributed Continuous Quality Assurance Process for Evaluating QoS of Performance Intensive Software, Studia Informatica Universalis, Volume 4 March 2005.

2. Arvind S. Krishna, Nanbor Wang, Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt and Gautam Thaker, CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations, The International Journal of Time-Critical Computing Systems, Springer, Vol. 29, Nos. 2-3, March-April 2005.

3. Arvind S. Krishna, Cemal Yilmaz, Atif Memon, Adam Porter, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan, Preserving Distributed Systems Critical Properties: A Model-Driven Approach, the IEEE Software special issue on the Persistent Software Attributes, Nov/Dec 2004.

4. Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt, Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems, International Journal of Embedded Systems, special issue on Design and Verification of Real-Time Embedded Software, April 2005.

5. Cemal Yilmaz, Adam Porter, Atif Memon, Arvind S. Krishna, Douglas C. Schmidt, and Aniruddha Gokhale, Techniques and Processes for Improving the Quality and Performance of Open-Source Software, Software Process - Improvement and Practice Journal: Special Issue on Free/Open Source Software Processes, 2006.

6. Aniruddha Gokhale, Krishnakumar Balasubramanian, Arvind S. Krishna, Jaiganesh Balasubramanian, and George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt, Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications, Elsevier Journal of Science of Computer Programming: Special Issue on Model Driven Architecture, Edited by Mehmet Aksit, 2004.

Referred Conference Publications

1. Arvind S. Krishna, Aniruddha Gokhale, Douglas C. Schmidt, John Hatcliff, and Venkatesh Prasad Ranganat, Towards Highly Optimized Real-time Middleware for Software Product-line Architectures, Proceedings of the 26th IEEE Real-time Systems Symposium (RTSS), Work in Progress Session, Miami, Florida, Dec 5-8, 2005

2. Arvind S. Krishna, Douglas C. Schmidt, and Michael Stal Context Object: A Design Pattern for Efficient Middleware Request Processing, Proceedings of the 12th Pattern Language of Programming Conference, Allerton Park, Illinois, September 7-10, 2005.

3. Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt, Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems, Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, CA, March 2005.

4. Cemal Yilmaz, Arvind S. Krishna, Atif Memon, Adam Porter, Douglas C. Schmidt, Aniruddha Gokhale, and Bala Natarajan, Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems, proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, May 15-21, 2005.

5. Arvind Krishna, Douglas C. Schmidt, Adam Porter, Atif Memon, Diego Sevilla-Ruiz, Improving the Quality of Performance-intensive Software via Model-integrated Distributed Continuous Quality Assurance, The 8th International Conference on Software Reuse, ACM/IEEE, Madrid, Spain, July 2004.

6. Arvind S. Krishna, Nanbor Wang, Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt and Gautam Thaker, CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations, Proceedings of the 10th IEEE Real-time Technology and Application Symposium (RTAS '04), Toronto, CA, May 2004.

7. Arvind S. Krishna, Douglas C. Schmidt, and Raymond Klefstad, Enhancing Real-Time CORBA via Real-Time Java, Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS), March 23-26, 2004, Tokyo, Japan.

8. Arvind S. Krishna, Douglas C. Schmidt, Krishna Raman, and Raymond Klefstad, Enhancing Real-time CORBA Predictability and Performance, Proceedings of the Proceedings of the 5th International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, November 2003.

9. Arvind S. Krishna, Douglas C. Schmidt, Raymond Klefstad, and Angelo Corsaro, Towards Predictable Real-time Java Object Request Brokers, Proceedings

of the 9th IEEE Real-time/Embedded Technology and Applications Symposium (RTAS), Washington DC, May 27-30, 2003.

10. Raymond Klefstad, Arvind S. Krishna, and Douglas C. Schmidt, Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, Embedded CORBA Applications, Proceedings of the Distributed Objects and Applications (DOA) conference, Irvine, CA, October/November, 2002.

Referred Workshop Publications

1. Arvind S. Krishna, Aniruddha Gokhale, Douglas C. Schmidt, Venkatesh Prasad Ranganath, and John Hatcliff, Model-driven Middleware Specialization Techniques for Software Product-line Architectures in Distributed Real-time and Embedded Systems, MODELS 2005 workshop on MDD for Software Product-lines: Fact or Fiction?, October 2, 2005, Jamaica.

2. Arvind S. Krishna Enhancing Middleware Quality of Service, 19th ACM OOPSLA Student Research Competition, Vancouver, Canada, Oct 2004.

3. Arvind S. Krishna, Emre Turkay, Cemal Yilmaz, Douglas C. Schmidt, Aniruddha Gokhale, Atif Memon and Adam Porter, Model-driven Software Tools for Configuring and Customizing Middleware for Distributed Real-time and Embedded Systems, 19th ACM OOPSLA Workshop on Managing Variabilities Consistently in Design and Code, Vancouver, Canada, Oct 2004.

4. Arvind S. Krishna, Cemal Yilmaz, Atif Memon, Adam Porter, Douglas C. Schmidt, Aniruddha Gokhale and Balachandran Natarajan, A Distributed Continuous Quality Assurance Process to Manage Variability in Performance-intensive Software, 19th ACM OOPSLA Workshop on Component and Middleware Performance, Vancouver, Canada, Oct 2004.

5. Cemal Yilmaz, Arvind S. Krishna, Atif Memon, Adam Porter, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan, A Model-based Distributed Continuous Quality Assurance Process to Enhance the Quality of Service of Evolving Performance-intensive Software Systems, Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS), Edinburgh, Scotland, UK, May 24, 2004

6. Arvind S. Krishna, Jaiganesh Balasubramanian, Aniruddha Gokhale, Douglas C. Schmidt, Diego Sevilla, Gautam Thaker, Empirically Evaluating CORBA Component Model Implementations, Proceedings of the ACM OOPSLA 2003 Workshop on Middleware Benchmarking, Anaheim, CA, October 26, 2003.

BIBLIOGRAPHY

[1] S.M. Abramov and N.V. Kondratjev. A Compiler Based on Partial Evaluation. In *Problems of Applied Mathematics and Software Systems*, pages 66–69. Moscow State University, Moscow, USSR, 1982. (In Russian).

[2] P.H. Andersen. Partial Evaluation Applied to Ray Tracing. DIKU Research Report 95/2, DIKU, 1995.

[3] Mary L. Bailey, Burra Gopal, Prasenjit Sarkar, Michael A. Pagels, and Larry L. Peterson. PathFinder: A Pattern-Based Packet Classifier. In *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*. USENIX Association, November 1994.

[4] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 190–199, San Francisco, CA, March 2005. IEEE.

[5] Krishnakumar Balasubramanian, Douglas C. Schmidt, Nanbor Wang, and Christopher D. Gill. Towards Composable Distributed Real-time and Embedded Software. In *Proc. of the* 8^{th} *Workshop on Object-oriented Real-time Dependable Systems*, Guadalajara, Mexico, January 2003. IEEE.

[6] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-time Specification for Java*. Addison-Wesley, 2000.

[7] Donald F. Box, Douglas C. Schmidt, and Tatsuya Suda. ADAPTIVE: An Object-Oriented Framework for Flexible and Adaptive Communication Protocols. In *Proceedings of the* 4^{th} *IFIP Conference on High Performance Networking*, pages 367–382, Liege, Belgium, 1993. IFIP.

[8] L. Breiman, J. Freidman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterey, CA, 1984.

[9] Gerald Brose, Nicolas Noffke, and Sebastian Müller. JacORB 1.4 Programming Guide. `jacorb.inf.fu-berlin.de/ftp/doc/ProgrammingGuide_1.4.pdf`, 2001.

[10] Darrell Brunsch, Carlos O'Ryan, and Douglas C. Schmidt. Designing an Efficient and Scalable Server-side Asynchrony Model for CORBA. In *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, Snowbird, Utah, June 2001. ACM SIGPLAN.

[11] Bruce Childers, J. Davidson, and M.L Soffa. Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.

[12] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[13] James Coplien, Daniel Hoffman, and David Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6), November/December 1998.

[14] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[15] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[16] Gary 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, August 2004. ACM.

[17] Mayur Deshpande, Douglas C. Schmidt, Carlos O'Ryan, and Darrell Brunsch. Design and Performance of Asynchronous Method Handling for CORBA. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications*, Irvine, CA, October/November 2002. OMG.

[18] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox, Jr. The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques. In *Proceedings of INFOCOM '97*, pages 179–187, Kobe, Japan, April 1997. IEEE.

[19] Bryan S. Doerr and David C. Sharp. Freeing Product Line Architectures from Execution Dependencies. In *Proceedings of the 11th Annual Software Technology Conference*, April 1999.

[20] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the* 13^{th} *Symposium on Operating System Principles*, pages 122–136, Pacific Grove, CA, October 1991. ACM.

[21] E. Ruf and D. Weise. Opportunities for Online Partial Evaluation. Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA, April 1992.

[22] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, pages 53–59, Stanford

University, California, USA, August 1996. ACM Press.

[23] David C. Feldmeier. Multiplexing Issues in Communications System Design. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 209–219, Philadelphia, PA, September 1990. ACM.

[24] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, 1999.

[25] Free Software Foundation. GCC Home Page. `gcc.gnu.org`, 2003.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[27] Christopher Gill, Douglas C. Schmidt, and Ron Cytron. Multi-Paradigm Scheduling for Distributed Real-time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), January 2003.

[28] Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind S. Krishna, George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2005 (to appear).

[29] Aniruddha Gokhale and Douglas C. Schmidt. Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA. In *Proceedings of GLOBECOM '97*, Phoenix, AZ, November 1997. IEEE.

[30] Aniruddha Gokhale, Douglas C. Schmidt, Balachandra Natarajan, and Nanbor Wang. Applying Model-Integrated Computing to Component Middleware and Enterprise Applications. *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, 45(10), October 2002.

[31] J. Grundy, Y. Cai, and A. Liu. Generation of Distributed System Test-beds from High-level Software Architecture Description. In *16 ^{t}h International Conference on Automated Software Engineering, Linz Austria*. IEEE, September 2001.

[32] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[33] John 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.

[34] John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, and Venkatesh Prasad. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[35] James Hu and Douglas C. Schmidt. JAWS: A Framework for High Performance Web Servers. In Mohamed Fayad and Ralph Johnson, editors, *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. Wiley & Sons, New York, 1999.

[36] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

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

[38] Mahesh Jayaram and Ron Cytron. Efficient Demultiplexing of Network Packets by Automatic Parsing. In *Proceedings of the Workshop on Compiler Support for System Software (WCSSS 96)*, University of Arizona, Tucson, AZ, February 1996.

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

[40] 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.

[41] Arvind S. Krishna, Douglas C. Schmidt, Raymond Klefstad, and Angelo Corsaro. Towards Predictable Real-time Java Object Request Brokers. In *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS)*, Washington, DC, May 2003. IEEE.

[42] Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt. Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 180–189, San Francisco, CA, March 2005. IEEE.

[43] Arvind S. Krishna, Cemal Yilmaz, Adam Porter, Atif Memon, Douglas C. Schmidt, and Aniruddha Gokhale. Distributed Continuous Quality Assurance Process for Evaluating QoS of Performance Intensive Software. *Studia Infomatica Universalis*, 4, March 2005.

[44] John Lakos. *Large-scale Software Development with C++.* Addison-Wesley, Reading, Massachusetts, 1995.

[45] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[46] L. Lei, G.-H. Moll, and J. Kouloumdjian. A Deductive Database Architecture Based on Partial Evaluation. *SIGMOD Record*, 19(3):24–29, September 1990.

[47] DM Levine, P.P Ramsey, and R.K Schmidt. *Applied Statistics for Engineers and Scientists: using Microsoft Excel and MINITAB.* Prentice Hall, 2001.

[48] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, Massachusetts, 1997.

[49] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr. Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 625–634. IEEE, April 2001.

[50] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Feedback Control Real-time Scheduling: Framework, Modeling, and Algorithms. *Real-time Systems Journal*, 23(1/2):85–126, July 2002.

[51] Silvano Maffeis. The Object Group Design Pattern. In *Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies*, Toronto, Canada, June 1996. USENIX.

[52] Renaud Marlet, Scott Thibault, and Charles Consel. Efficient Implementations of Software Architectures via Partial Evaluation. *Automated Software Engineering: An International Journal*, 6(4):411–440, October 1999. Available from: `citeseer.csail.mit.edu/marlet99efficient.html`.

[53] Mats Bjorkman and Per Gunningberg. Locking Strategies in Multiprocessor Implementations of Protocols. *Transactions on Networking*, 3(6), 1996.

[54] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Conference*, pages 259–270, San Diego, CA, January 1993.

[55] Atif Memon, Adam Porter, Cemal Yilmaz, Adithya Nagarajan, Douglas C. Schmidt, and Bala Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

[56] Microsoft Corporation. Microsoft .NET Development. msdn.microsoft.com/net/, 2002.

[57] Jeffrey C. Mogul, Richard F. Rashid, and Michal J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *Proceedings of the* 11^{th} *Symposium on Operating System Principles (SOSP)*, November 1987.

[58] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of the* 1^{st} *Symposium on Operating Systems Design and Implementation*. USENIX Association, November 1994.

[59] S. Nimmagadda, C. Liyanaarnchchi, A. Gopinath, D. Niehaus, and A. Kaushal. Performance Patterns: Automated Scenario-Based ORB Performance Evaluation. In *Conference on Object Oriented Technologies and Systems*, pages 15–28, San Diego, CA, 1999.

[60] Object Management Group. *Lightweight CCM RFP*, realtime/02-11-27 edition, November 2002.

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

[62] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, December 2002.

[63] Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document ptc/03-07-08 edition, July 2003.

[64] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, Ossama Othman, and Jeff 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, April 2000.

[65] David L. Parnas. A Rational Design Process: How and Why to Fake it. *IEEE Transactions on Software Engineering*, February 1986.

[66] Adam Porter and Richard Selby. Empirically Guided Software Development Using Metric-Based Classification Trees. *IEEE Software*, March 1990.

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

[68] Calton Pu, Tito Autery, Andrew Black, Charles Consel, Crispin Cowan, Jonathan Walpole Jon Inouye, Lakshmi Kethana, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In

Symposium of Operating System Principles, Copper Mountain Resort, Colorado, December 1995.

[69] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha 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.

[70] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Using Principle Patterns to Optimize Real-time ORBs. *IEEE Concurrency Magazine*, 8(1), 2000.

[71] Irfan Pyarali and Douglas C. Schmidt. An Overview of the CORBA Portable Object Adapter. *ACM StandardView*, 6(1), March 1998.

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

[73] Tom Ritter, Marc Born, Thomas Unterschütz, and Torben Weis. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, Honolulu, HW, January 2003. HICSS.

[74] Richard J. Rodger. Jostraca: a Template Engine for Generative Programming. In *ECOOP 2002 Workshop on Generative Programming*. Springer Verlag, 2002.

[75] Wendy 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)*. IEEE/IFIP, May 2003.

[76] Matthew Rutherford and Alexander L. Wolf. A Case for Test-Code Generation in Model-Driven Systems. In *International Conference on Generative Programming and Component Engineering (GPCE) 2003, Erfurt Germany*. ACM SIGPLAN SIGSOFT, September 2003.

[77] C. Sakama and H. Itoh. Partial Evaluation of Queries in Deductive Databases. *New Generation Computing*, 6(2,3):249–258, 1988.

[78] Sunil Saxena, J. Kent Peacock, Fred Yang, Vijaya Verma, and Mohan Krishnan. Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes. In *Proceedings of the Winter USENIX Conference*, pages 85–106, San Diego, CA, January 1993.

[79] Richard E. Schantz and Douglas C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering*. Wiley & Sons, New York, 2002.

[80] Douglas Schmidt and Steve Vinoski. Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Object. *C++ Report*, 8(7), July 1996.

[81] Douglas Schmidt and Steve Vinoski. Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Request. *C++ Report*, 8(2), February 1996.

[82] Douglas Schmidt and Steve Vinoski. Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool. *C++ Report*, 8(4), April 1996.

[83] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. In *Proceedings of the* 2^{nd} *C++ Conference*, pages 87–102, San Francisco, California, April 1990. USENIX.

[84] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Boston, 2002.

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

[86] Douglas C. Schmidt, Vishal Kachroo, Yamuna Krishnamurthy, and Fred Kuhns. Applying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications. *IEEE Communications Magazine*, 38(10):112–123, October 2000.

[87] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[88] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

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

[90] David C. Sharp. Avionics Product Line Software Architecture Flow Policies. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 1999.

[91] David C. Sharp and Wendy 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.

[92] Monchai Sopitkamol and Daniel A. Menasce. A Method for Evaluating the Impact of Software Configuration Parameters on E-Commerce Sites. In *Proceedings of the 2005 Workshop on Software and Performance*, Palma de Mallorca, Spain, July 2005.

[93] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.

[94] Sun Microsystems. Java^{TM} 2 Platform Enterprise Edition. java.sun.com/j2ee/index.html, 2001.

[95] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.

[96] Clemens Szyperski. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley, Santa Fe, NM, 1998.

[97] David L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

[98] P. Thiemann and M. Sperber. Program Generation With Class. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik'97, Aachen, Germany, September 1997*. Berlin: Springer-Verlag, 1997.

[99] Todd Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–34, 1999.

[100] Andreas Vogel, Brett Gray, and Keith Duddy. Understanding any IDL-Lesson one: DCE and CORBA. In P. Honeyman, editor, *Proceedings of Second International Workshop on Services in Distributed and Networked Environments*, Los Alamitos, CA, 1996. IEEE Computer Society Press. In Press.

[101] Nanbor Wang, Christopher Gill, Douglas C. Schmidt, and Venkita Subramonian. Configuring Real-time Aspects in Component Middleware. In *Lecture Notes in Computer Science: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04)*, volume 3291, pages 1520–1537, Agia Napa, Cyprus, October 2004. Springer-Verlag.

[102] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill. QoS-enabled Middleware. In Qusay Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2003.

[103] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[104] M. Yuhara, B. Bershad, C. Maeda, and E. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the Winter Usenix Conference*, January 1994.

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

[106] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of the International Conference on Distributed Systems 2002*, July 2002.