Optimization Techniques for Enhancing Middleware Quality of Service for Product-line Architectures

Arvind S. Krishna
arvindk@dre.vanderbilt.edu
Electrical Engineering and Computer Science Dept
Vanderbilt University, Nashville, TN 37235

Abstract

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. The goal of PLAs is to reduce product development costs via systematic reuse, while enhancing key product quality factors, such as assured latency/jitter/throughput values, scalability, and dependability.

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 thesis proposal 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 proposal describes the OPTEML solution approach that synergistically addresses key unresolved research challenges via optimization strategies that encompass pattern-oriented, model-driven development, and partial specialization techniques to enhance the QoS and flexibility of middleware for PLAs. These optimizations will be prototyped, integrated, and validated in the context of several representative applications using middleware developed with Real-time Java and C++.

1 Introduction

1.1 Emerging Trends and Technologies

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) [10] and *middleware* [75] are promising technologies for addressing these demands.

1.1.1 Product-line Architectures

In contrast to conventional software processes that produce separate point solutions, in a PLA-based process, a family of product variants [84] is developed to share a common set of capabilities, patterns, and architectural styles. For example, Figure 1 illustrates a portion of the Boeing Bold Stroke avionics mission computing PLA [86], 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 [62] and heavily influenced by the Lightweight CORBA Component Model (CCM) [61, 70].

PLAs in general – and Bold Stroke in particular – can be characterized using the *Scope, Commonality, and Variabilities* (SCV) analysis [11]. 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).

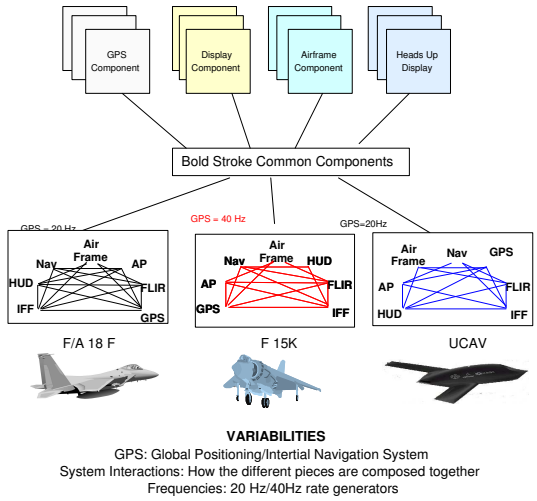

Figure 1: SCV Analysis for Boeing Bold Stroke PLA

After a PLA has been developed and has matured, the ensuing development of product variants ideally proceeds in top down manner. Figure 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 [28, 31] 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 [63], J2EE [89] or .NET [57].

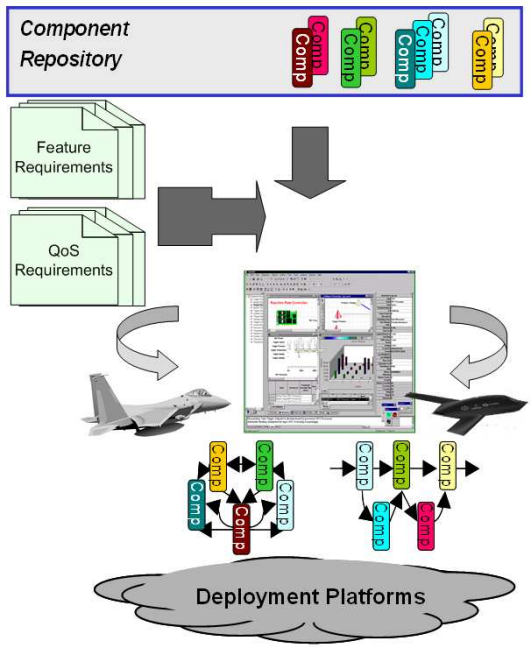

Figure 2: PLA Development Process

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* [17, 84, 86]. Examples of DRE systems include applications with hard real-time requirements, such as avionics mission computing [82], as well as those with softer real-time requirements, such as telecommunication call processing and streaming video [81]. 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.

1.1.2 Middleware

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 [75]. 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 3 illustrates a widely applied middleware architecture [25] that underlies PLAs used for DRE systems [25, 51, 84–86]. This figure illustrates two

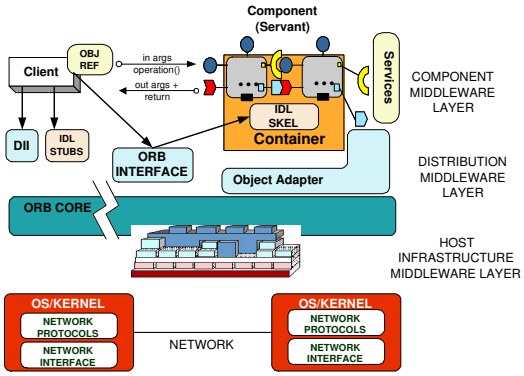

Figure 3: Layered Middleware Architecture

key characteristics of middleware:

- **Design for generality**, where each layer is designed to host different applications. For example, the Java Virtual Machine (JVM) [50] 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 [62] and Real-time Java [5], 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 [61] and Prism [84], 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.

1.2 Challenges with Using Existing Middleware Platforms for PLAs

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 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. Resolving this tension is essential to ensure

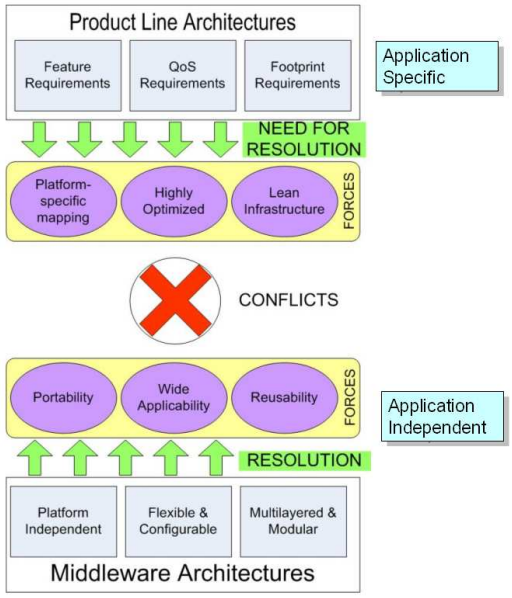

Figure 4: Application-specific vs. Application-independent Dimensions of PLAs and Middleware

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 [62] and Real-time Java [5] outlined in Section 1.1.2, 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 configuations 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.

1.3 Research Approach

To address the challenges described in Section 1.2, the proposed project will develop Optimization Techniques for Enhancing Middleware QoS for PLAs (OPTEML). The different dimensions of this approach are shown in Figure 5 and will:

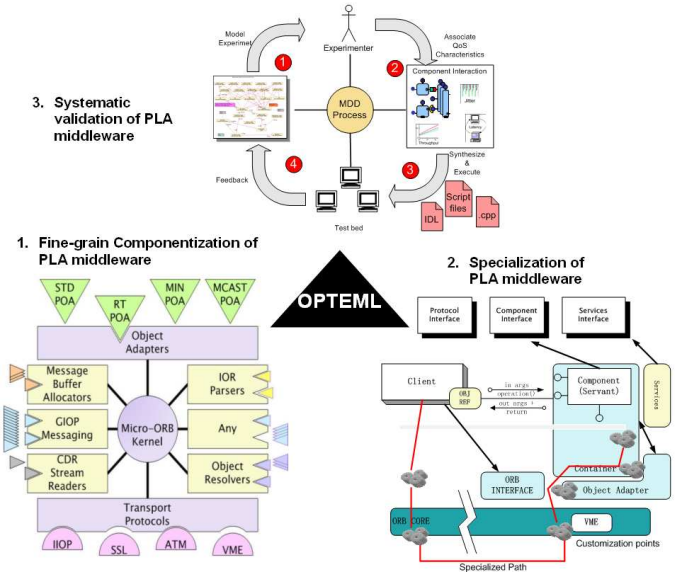

Figure 5: Dimensions of OPTEML Research

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 a monolithic

element and factored out into modular components at a finer level of granularity. The proposed research will examine the fine-grain componentization of the CORBA *Portable Object Adapter (POA)* [69] using a policy-driven approaches. Section 3.1 describes the proposed approach for fine-grained componentization of PLA middleware in detail.

2. Specialize PLA middleware to eliminate unnecessary time and space overhead. This approach will use a two step process to customize middleware. In the first step, the middleware source code will be annotated with customization hooks that can resolve to nothing when used in the general case. In the specialized case, an external tool will read specialization rules (stored in a rules file) and use the hooks to specialize the middleware source code. This approach is designed to eliminate middleware generality, such as redundant checks, that are not required for particular product variants. Section 3.2 describes the proposed 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 will be 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. Section 3.3 describes the proposed approach for validating PLA middleware configurations in detail.

1.4 Proposal Organization

The remainder of this proposal is organized as follows: Section 2 presents a taxonomy of existing research efforts that are related to the thesis proposal and uses this taxonomy to identify key challenges that have not been addressed adequately in existing research on PLAs; Section 3 describes how the research in this proposal will fill the gaps identified in Section 2; Section 4 provides a time-line for the thesis proposal and summarizes the contributions of this work.

2 Research Evolution

This section systematically explores and documents related research addressing different issues relating to the research focus on optimization techniques for enhancing middleware quality of service for PLA. To structure the discussion, a taxonomy, *i.e.*, classification, is presented that categorizes related research across the following two dimensions shown in Figure 6:

- **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.

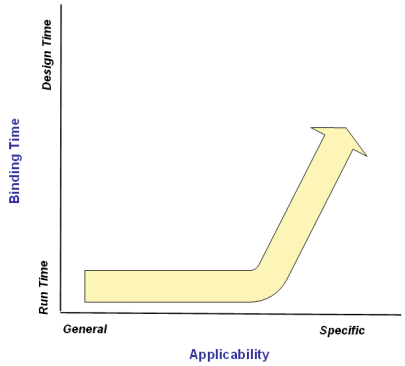

Figure 6: Research Taxonomy

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, and
- **Partial 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 section 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 requires resolution.

2.1 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 7, 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 request processing within QoS enabled middleware implementations. These address 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. In addition, these optimizations are applied at run-time and fall into our taxonomy as

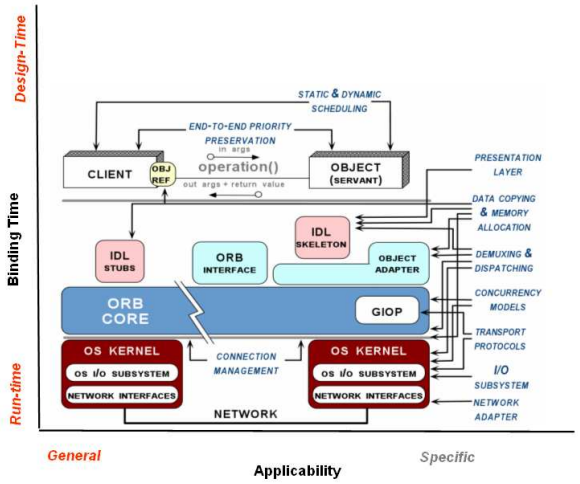

Figure 7: General-purpose Optimizations

general-purpose run-time optimization techniques.

2.1.1 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, [16,20,90] 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 [58]. A number of schemes to implement fast and efficient packet filters are available. These in-

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

Table 1: Dimensions of General-purpose Optimizations

clude the BSD Packet Filter (BPF) [55], the Mach Packet Filter (MPF) [97], PathFinder [3], demultiplexing based on automatic parsing [36], and the Dynamic Packet Filter (DPF) [19].

In the CORBA middleware, research efforts have focused on ensuring $O(1)$ demultiplexing time bound for different layers within CORBA middleware. Perfect hashing [79] 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 [30]), the objects and operations can be configured statically. Research effort in [27] has used perfect hashing to generate hash functions for operation names defined in IDL. In other efforts [68], 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 [54] 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; [59] 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 [74] 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 [6] framework examines research on parallelizing transport architectures. The JAWS [32] 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 [18] has looked into implementation of *continuations* in the MACH kernel, which decouples the request demultiplexing from the processing. In ORBs, Asynchronous Method Handling (AMH) [15] provide similar mechanisms like continuations. On the client side, the Asynchronous Method Invocation (AMI) [7] ameliorates clients blocking overheads when waiting for reply for a long running request.

Summary. The research efforts discussed earlier and related efforts documented in [76–78] to better implement high performance architectures have distilled into systematic body of knowledge in the form of design patterns [22]. The Pattern Oriented Software Architecture (POSA) 2 [83] 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 1 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 il-

lustrates the different domains on which these optimizations have been applied.

2.1.2 What Remains to be Done

Traditional general-purpose optimizations have looked at performance related issues of middleware. These have attained maturity and also have resulted in demonstrating 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 [47]. 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. Section 3.1 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 [33] 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. Section 2.2 describes how configuration-driven optimization techniques are addressing some of these issues.

2.2 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 tradeoff, in terms of application QoS, in terms of performance 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 [21], 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]

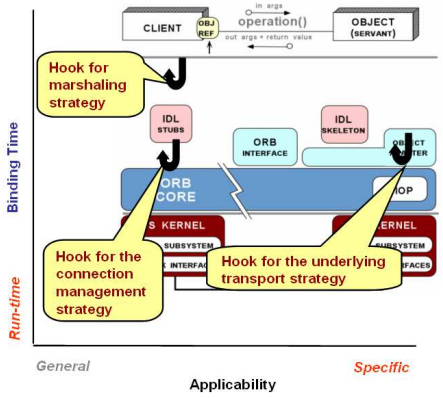

Figure 8: Configuration-driven Optimizations

Similar to how one needs to understand the consequences of enabling different compiler options, Figure 8 illustrates the need for product-line variants to understand the consequences of middleware set-

[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/`.

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

2.2.1 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 [38] 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 [98], 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 [52] 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 [9] 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 runtime 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 [72] 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 [29] 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 [56] is a distributed continuous quality assurance (DCQA) tool for developing and validating novel software QA processes and tools. Skoll lever-

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

Table 2: Dimensions of Configuration-driven Specialization Mechanisms

ages 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 [60] 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 [87] 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 [49] analysis to determine the statistical significance of different web-server options.

Summary. Table 2 categorizes configuration driven approaches along two dimensions. The first dimension describes 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.

2.2.2 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 [44, 46] identified how tedious and error-prone the process of evaluating these configurations really were. Section 3.2 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 implementation. Section 2.3 discusses research approaches that address this issue.

2.3 Partial Specialization Optimizations

Jones et al. [37], define Partial Specialization (partial 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 [35], compilers [1], program generation [91] and generative programming techniques [13]. Partial specialization 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 specific. Figure 9 illustrates how AOT properties can be used to specialize middleware.

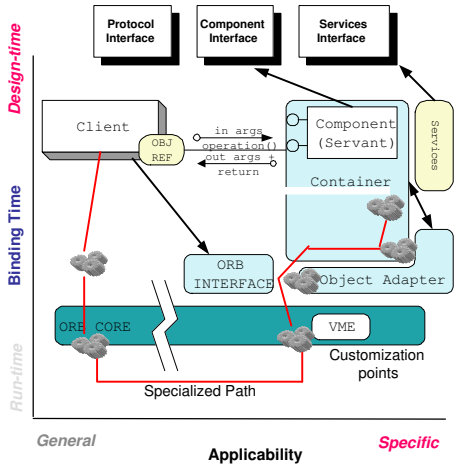

Figure 9: Partial Specialization Optimizations

2.3.1 Dimensions of Partial Specialization Mechanisms

This section analyzes different dimensions of partial specialization 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. Partial 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 partial 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 [71].

Language mechanisms. Common partial specialization mechanisms at the language level include approaches that are two level in nature as described in [93]. 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 [94] and rpcgen [88] (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 partial specialization leverages language level partial 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 partial 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 partial specialization techniques. Partial specialization mechanisms have been applied to different domains including scientific applications, functional programming, operating systems and database systems. This section provides examples of partial 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 [73], general purpose queries have been transformed into specific programs optimized for a given input. Similarly, training neural networks [48] for a given scenario has improved its performance.

In addition to the aforementioned domains, partial 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 procedure calls. Others have extended this approach to use incremental specialization techniques. For example in their work [66], 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 3 categorizes partial specialization initiatives across three different dimensions. The first dimension shows the different partial 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 spe-

Partial Evaluation & 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	*partial specialization tools*
Lambda Calculus	Lamdamix
Prolog	Logimix
Scheme	Similix
C	C-mix

Table 3: Dimensions of different partial specialization mechanisms

cialize programs written in the corresponding languages. We do not discuss each of the tools in detail, more information about each tool is available in [37].

2.3.2 What Remains to be Done

Traditional partial specialization techniques have been used to optimize applications in function/logic programming. There does not exist any partial specialization tool for object oriented programming languages such as C++ or Java. partial 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 [42] 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 accomodate 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. Section 3.2 how this limitation is resolved using the OPTEML approach.

2.4 Summary

The research evolution and taxonomy presented in this section 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 section 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, and

The remaining portion of this proposal describes how the aforementioned research challenges will be addressed systematically.

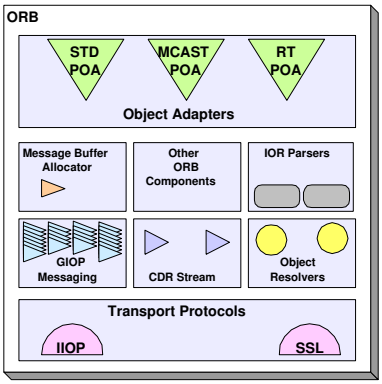

Figure 10: Monolithic ORB Architecture

3 Thesis Proposal

This section describes a proposal for resolving research challenges identified in Section 2.3.2 that remain unresolved by the related work described in Section 2

3.1 Challenge 1: Fine-grain Middleware Componentization complexities

The taxonomy on general-purpose optimizations (described in Section 2.1.2) identified limitations with existing middleware architectures for enabling fine-grain customizability of middleware features. This section describes how this shortcoming is addressed in OPTEML.

Context. 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.

Research Challenges. Implementing a full-service, flexible, specification-compliant ORB can yield a monolithic ORB implementation with a large memory footprint as shown in Figure 10. Moreover, the footprint grows with each extension (adding support for a new protocol), and extensibility is hard. A monolithic ORB, however, does not suffer from any indirection overheads example indirection cost from virtual functions. Therefore the research challenges of developing a feature rich extensible middleware implementation can be broken down into the following sub-problems: (1) Enabling middleware to provide a rich and configurable set of functionality, yet be configurable to incur footprint (main memory) only for components that are actually used; (2) Making the ORB easily extensible in order to facilitate the development of different alternatives; and (3) Allowing both *static* and *dynamic* configuration, to allow the application developer to choose a trade off between maximal efficiency and flexibility and (4) Ensuring end-to-end deadline and performance requirements.

Hypotheses. This proposal explores and validates the following hypotheses: The solution approach (1) enables 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), (2) is transparent to the application (no changes to the PLA application code), (3) significantly reduces middleware footprint and (4) allows easy addition of new features without sacrificing performance.

Solution Approach → Micro-ORB Architectures. 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) Identifies 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 [25] 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. Provide a factory [22], for example, a protocol factory that creates different protocols depending on configuration settings.

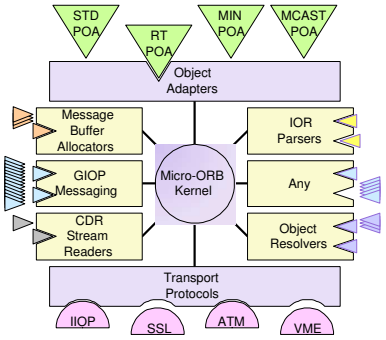

Figure 11: Micro ORB Architecture

Figure 11 illustrates how the application of the aforementioned approach results in the factoring of ORB services from the core to the periphery. Using this approach, principal core ORB services (object adapters, message buffer allocators, GIOP message handling, CDR Stream readers/writers, protocol transports, object resolvers, IOR parsers, and `Any` handlers) are moved out of the ORB to reduce its memory footprint and increase its flexibility. Each ORB service itself is decomposed into smaller pluggable components that can be loaded into the ORB only when needed.

This approach provides a *coarse-grain* solution where the component as a whole is either plugged in or removed wholesale. For example, "pure clients" do not need the Portable Object Adapter (POA) [69] component. 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. This approach however does not enable further componentization of the POA for servers which do need this component.

In a fine-grain approach, the POA component is considered as a monolithic piece and further decomposed to enhance customizability. The POA provides several policies to customize its behavior. For example, the Life time POA policy dictates whether a POA is persistent or transient. Each policy defines different behaviors out of which only one can be associated during the lifetime of a POA. It is therefore possible to decompose the POA along policy alternatives. In this aggressive approach, rather than an all or nothing solution, individual policy alternative components can be plugged in or out as needed. For example, Figure 12 shows the fine-grain architecture of the ZEN[3] POA, where each POA policy is factored out into a separate class hierarchy by applying the Strategy pattern [22].

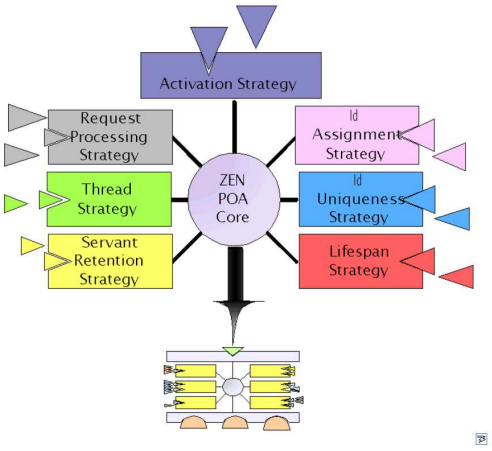

Figure 12: Fine-grain Architecture applied to the ZEN POA

Each hierarchy has an abstract *base strategy* and *concrete strategies* corresponding to the policy values

[3]ZEN [8] is an open-source implementation of Real-time CORBA using the Real-Time Specification for Java (RTSJ).

associated with that policy. Further, many of these concrete strategies do not maintain any state and have been implemented using Flyweight pattern [22]. This pattern uses sharing to support large numbers of fine-grain objects efficiently, which means there is only one instance of the strategy. The results of application of our approach show that a fine-grained design can be half [39] the size of a monolithic implementation and yet be simultaneously predictable [40] and high-performance [42].

At the base of OPTEML is the aforementioned flexible middleware architecture that allows feature subsetting and minimizes the middleware implementation footprint. This architecture is based on the Virtual Component Pattern [12], that allows an application transparent way of loading and unloading components. This flexible architecture, has been implemented and validated using the ZEN ORB. Mature Real-time CORBA implementations, such as the ACE+TAO [33, 34], are embracing the fine-grain componentization approach to building ORB implementations. Overt this flexible architecture are novel optimization strategies, patterns, and idioms that help enhance QoS of DRE systems and facilitate the application of promising new technologies (such as the RTSJ) to DRE systems. Technology transitions and successes of TAO and ZEN have been documented in [41].

Validating the hypotheses. The fine-grain componentization approach validates the hypotheses detailed earlier by providing an application transparent approach for minimizing middleware footprint. Further, this approach neither requires any changes to existing PLA application code nor modifies the signature of the CORBA interfaces. The middleware implicitly figures out which components are needed based on policies defined or until components are resolved for use (via the `resolve_initial_reference()` operation). The coarse-grain and fine-grain approaches provide a tradeoff between different levels of *middleware customization.*. This approach leads to *substantial improvement in footprint*, where a fine-grain architecture can be half the size of its monolithic couterpart. Finally, this approach *facilitates component extensibility* based on PLA requirements by localizing change within the particular component hierarchy. For example, to support a new protocol, middleware developers need to provide a new implementation conforming to the interface exposed by the base protocol strategy component. The base strategy then is the only component that needs to be modified to load/unload this new protocol.

3.2 Challenge 2: Middleware Specialization Complexities

The taxonomy on partial specialization optimizations identified limitations (described in Section 2.3.2) with existing partial specialization approaches. This section describes how this shortcoming is addressed in OPTEML.

Context. DRE system infrastructure continues to expand in scope and capabilities as new protocols and mechanisms are defined at the network, OS, and middleware layers. For example, middleware provides various protocol implementations [64] tailored towards changing operation contexts, *e.g.*, the Internet Inter-ORB Protocol (IIOP) can be used for web-based communication, whereas the Stream Control Transmission Protocol (SCTP) [24] can be used for fault-tolerant DRE applications.

The resolution of Challenge 1 enables middleware to implement the different protocol implementations, *i.e.*, IIOP and SCTP as pluggable alternatives to a generic protocol component. This ensures that a variant not requiring SCTP support does not incur the footprint overhead for supporting this protocol. This approach, however, cannot eliminate middleware generality for example, the generality in the middleware for providing plug and play of multiple protocol implementations.

Research challenges. The broader research challenge is to identify specialization points in the middleware to apply specialization techniques. If the specializations are applied at points that do not produce improvements in end-to-end performance and QoS then such approaches are counter-productive.

A second related challenge is that manually customizing middleware implementations, *i.e.*, by manually rewriting source code, is *ad hoc* and involves significant and error-prone changes to implementation code. Research challenges thus involve developing software environments that evaluate system properties (such as concurrency mechanisms and type of protocol implementation that are fixed at *design* time) and use this information to customize middleware implementations and eliminate additional layers of indirection.

Hypotheses. This proposal explores and validates the following hypotheses: The solution approach (1) identifies the specialization points in the middleware that lead to increased performance applicable to different PLA variants. (2) does not cause additional run-time performance overhead for the specialized version over and above the general-purpose version. (3) provides a framework for the addition of new specializations and (4) improves performance considerably for different product-line variants.

Solution Approach → Middleware Specialization Techniques. Partial specialization is an inter-procedural constant propagation technique that can improve performance and footprint by (1) tailoring services to the specific needs of software systems and (2) bypassing layers of abstraction to call directly to underlying platform protocols and mechanisms [53].

To customize middleware for different operating contexts, this research explores techniques for customizing middleware based on Ahead of Time (AOT) known system properties to improve middleware performance and footprint. In particular, this research proposes: refactoring middleware and services so they are amenable to automatic and dynamic customization and optimization by using middleware specialization patterns [14]. These patterns are then mapped to middleware implementations via specialization tools. The specialization process can be divided into two steps: (1) identification of the specialization points and transformations and (2) automating the delivery of the specializations.

The following are some of the possible road blocks in the specialization process that this research has identified:

- Maturity and availability of tools that can help in automatic specialization of middleware. For example, AspectJ is a mature implementation in Java where as Aspect C++ is yet to mature,
- Overheads imposed by tools might impede the target performance improvements. For example, the overhead of AspectJ runtime infrastructure, both in terms of linking to the library and run-time overhead might cause performance degradation as detailed in [23],
- Need for the specialized code and general-purpose code to exist simultaneously, and
- Compatibility issues such as compliance with the CORBA specification.

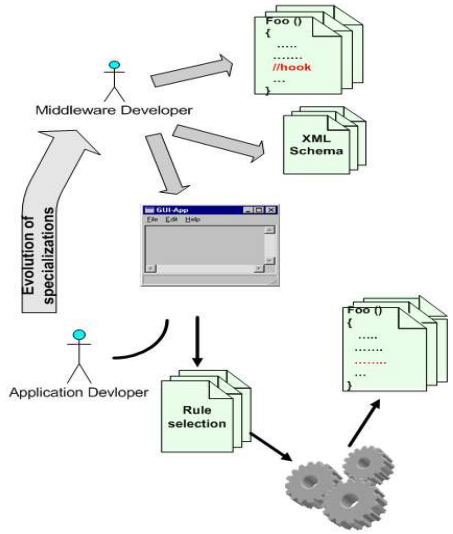

Figure 13: FOCUS Approach

To address the aforementioned limitation, this research is developing the FOCUS (Feature-oriented Customizer) tool. Figure 13 illustrates the actors, tools and workflow in FOCUS. The different phases in the FOCUS approach can be broken down as follows:

• **Identification of specializations.** The most important step in the specialization process is the identification of specialization points to eliminate generality. Rather than selecting ad-hoc points, this

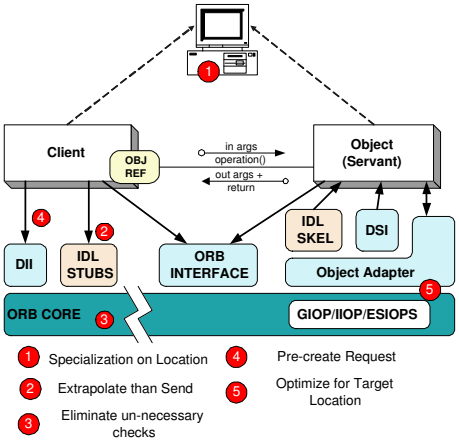

Figure 14: FOCUS: Specialization Identification

research will examine the critical request/response processing path within middleware to systematically identify sources for specialization. Identification of these points have the greatest potential for increasing performance improvement. Figure 14 illustrates the specialization points along the critical request/response that have been chosen for specialization including:

- **Specialization for target location,** which removes redundant code and checks along the request processing path when both client and server are collocated, *i.e.*, present in the same address space,
- **Specialization for request type,** which pre-creates an entire request when the same operation is invoked repeatedly by the client,
- **Once per state resolution of dispatch,** which bypasses middleware processing layers at the server side when the same operation is invoked along a given connection,
- **Extrapolate rather than send,** which first tries to obtain the result locally at the client before making a remote call, and
- **Eliminate demarshaling checks**, which eliminates redundant byte-order checks along the request processing path at the server.

In addition to the path specializations, component specializations eliminate indirection overheads (generality) in middleware component along the critical/request response processing path including the pluggable protocol and reactor [80] frameworks.

- **Capturing specializations as rules.** Figure 15 illustrates how specializations are expressed as rules. In this phase, a middleware developer lays down the specialization rule required to transform general-purpose middleware into optimized middleware stack. These directly stem from the specialization points identified in the previous step.

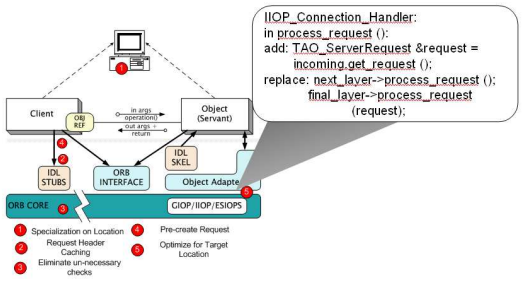

Figure 15: FOCUS: Capturing Specialization Rules

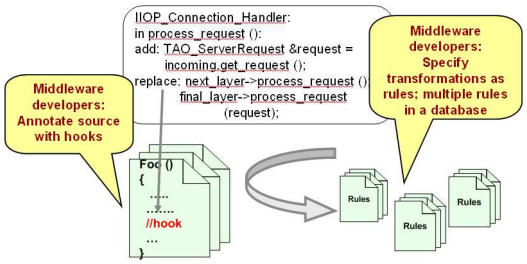

Figure 16: FOCUS: Middleware Annotation

- **Middleware Annotation.** Figure 16 shows how the rules are used to annotate the middleware. In conjunction with capturing the specialization rules, the middleware developer annotates the middleware source with specialization hooks. These hooks are inserted as comments in the source code that do not interfere with the normal request/response processing. However, in the specialized mode, these hooks are used to weave in specialized code using a customizer engine.
- **FOCUS Transformations.** Figure 17 illustrates the different steps in this phase. (1) A product-line application developer choses the specializations that are suitable for the variant. This

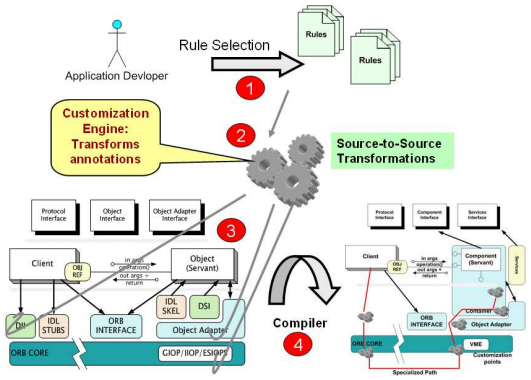

Figure 17: FOCUS: Transformation Process

process is manual or a GUI wizard is provided to infer specialization rules to be applied. (2) A transformation engine, then uses the rules specified in the rules file and (3) performs the transformations specified in the file using the hooks left in the middleware code. (4) Optimizing compiler then uses the modified source file to generate executable platform code.

The advantage of this approach is that the transformations are applied to the source directly enabling an optimizing compiler to generate optimized code. However, a disadvantage is that Aspects provide a superior mechanism of weaving code. However, irrespective of the actual approach, our specialization rules are applicable across different mechanisms. The need for generalized and special purpose code to coexist, introduces additional replugging cost when a specialized version needs to also support general-purpose behavior. Replugging costs can introduce considerable jitter for real-time systems. For this approach therefore avoids replugging costs by raising exceptions when the special purpose code also needs the general purpose behavior. The specialization in addition will not change the CORBA interfaces CORBA interfaces, for example, no addition of operation parameters to interface operations, which will avoid breaking CORBA compatibility.

Validating the hypotheses. The middleware specialization approach addresses hypotheses outlined earlier by (1) systematically identifying specializations along critical request/response path to ensure improvement in performance. (2) Providing a source to source transformation requiring no runtime overhand such as linking/calling into an external library. (3) Enabling the addition of new specializations as rules that can be added to the existing set of rules in the data base. If external tools like Aspects mature, the rules that capture the transformations can be re-written as aspects. Finally, the benefits to the application depends on how many of the specializations are enabled along the critical request/response path. Turning on all the specializations would enable ∼40% improvement in performance, while turning on just a couple of optimizations would improve performance by only ∼15%.

3.3 Challenge 3: Middleware Configuration Validation and Tuning Complexities

The taxonomy on configuration-driven optimizations (described in Section 2.2.2) identified limitations with ad hoc approaches to right middleware configuration selection & validation for different PLAs. This section describes how this shortcoming is addressed in OPTEML.

Context. 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.

Research challenges. 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.

Hypotheses. This proposal explores and validates the following hypotheses: The solution approach (1) eliminates accidental complexities in generation of scaffolding code. (2) enables validation of middleware configurations across diverse platforms OS and compiler settings, and (3) can be applied to identify middleware configurations influencing end-to-end performance/latency/jitter the most ("main effects") of a given PLA variant.

Solution Approach → Model Driven Distributed Continuous Quality Assurance Processes. To specifically address the repeatability, cost and automation limitations of middleware configuration tuning approaches discussed in Section 2.2.2, this research synergistically combines Model-Driven Development (MDD) techniques with Quality Assurance (QA) approaches. An MDD approach (for example the CoSMIC [26] 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 [56] enables the validation of the different middleware configurations for functional correctness and performance across diverse platforms by executing QA tasks continuously and intelligently.

This research proposes the validation of the solution approach using the development of a Domain Specific Modeling Language (DSML)s using GME to capture QoS evaluation concerns of different product-line variants. This tool will be integrated with the CoSMIC toolsuite to work with the (1) the Options Configuration Modeling language (OCML) [92] that allows developers to model middleware configuration options as high-level models and (2) the Platform Independent Component Modeling Language (PICML) [4] that captures component interactions in higher level models and synthesizes XML-metadata to deploy the component system. The generated code from the tool will be integrated with the Skoll framework (`http://www.cs.umd.edu/projects/skoll/`), which is a prototype DCQA environment to validate and document how middleware configurations affect product-line QoS across a range of platforms OS and compiler settings.

The research contribution for this dimension of OPTEML has been the development of the Benchmarking Generation Modeling Language (BGML) [43], a DSML to evaluate the QoS of different middleware configurations. BGML provides the capability of modeling different types of operations (one way, two way), QoS metrics (latency, throughput) and background load (continuous, rate based or interactive) information and generating scaffolding to evaluate middleware QoS. BGML has been integrated with the CoSMIC tool-suite and works in concert with OCML and PICML DSMLs. Figure 18 visually illustrates how BGML and OCML work in concert to automate the generation of scaffolding code required evaluating product-line QoS.

The second contribution is the development of a Model-Driven DCQA processes for middleware configuration validation. Figure 19 visually depicts the high level steps involved in this process and described below:

- **Step 1: Define product-line scenario.** An application developer uses CoSMIC to define the scenario, choose a middleware configuration space and model the QoS requirements such as end-to-end latencies.
- **Step 2: Generate scaffolding code.** In this step, the model interpreters are used to generate all the scaffolding code required to enact the process. This step also generates the middleware configuration files, which is then fed to Skoll's configuration navigation agent.
- **Step 3: Register and download clients.**

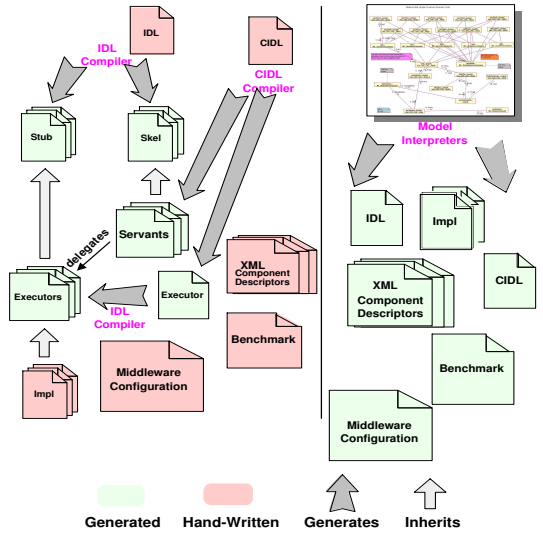

Figure 18: A Comparison of Handcrafted and Generated Code

Remote users register with the Skoll infrastructure and obtain the Skoll client software and code artifacts to run.

- **Step 4: Execute.** Skoll software runs the benchmarks on different configurations to evaluate QoS and documents the variation of QoS.

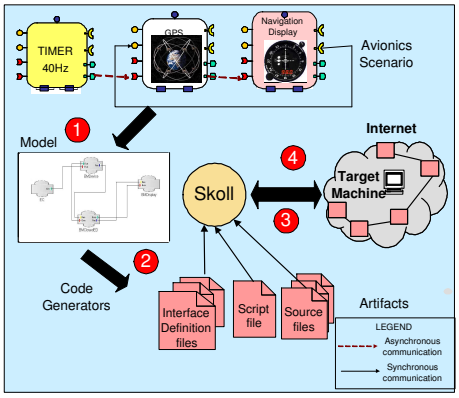

Figure 19: Model Driven DCQA Approach

This tool integration process has been applied to a real-life product-line scenario to determine the right set of configurations satisfying end-to-end QoS [44].

Finally, the Model-driven DCQA process has been applied to estimate configuration "main effects" *i.e.*, subset of options that account for the greatest portion of performance variations across the system's configuration space. In this approach rather than exhaustively testing the configuration space, which is expensive and time consuming, a screening design [95] technique is used. This approach is highly economical and can reveal important *low order effects* (such as individual option settings and option pairs/triples) that strongly affect performance. These configurations are "main effects." The marriage of MDD and DCQA techniques enables automation, repeatability and validation of configuration main-effects for different PLA variants across a range of platforms, OS and compiler settings as discussed in [96]. This process has also been generalized [45] to be applicable to preserve Persistent Software Attributes (PSA)s such as portability, functional correctness and reliability.

Validating the hypotheses. The Model-Driven DCQA processes address the hypotheses outlined earlier in the following manner: (1) The tool-integration process eliminates accidental complexities in generation of all the scaffolding code required to enact a PLA scenario. (2) The integration of Skoll, BGML and OCML enables validation of middleware configurations of different PLAs across diverse platforms OS and compiler settings. (3) The main-effects screening DCQA process provides a reusable and automatable solution that can be applied to identify principal configuration settings affecting end-to-end QoS such as performance/latency and jitter measurements.

4 Proposal Timeline & Summary of Research Contributions

Figure 20 shows research progress and a plan for the completion of OPTEML. The first phase of OPTEML completed as of May 2003, involved the development of patterns for the fine-grain componentization of CORBA ORB middleware. The fine-grain componentization efforts have been implemented and validated in both ZEN and TAO. The second phase of OPTEML (May 2003 – Nov 2004)

OPTEML: Research Publications	
Fine-grain Componentization	*Publications & acceptance rate*
POA architecture ORB-core architecture Real-time performance	Distributed Objects and Applications (DOA) 2003 (25%) DOA 2003 (30%) Real-time Application Symposium (RTAS) 2004 (25%) International Conference on Distributed Systems (ICDCS) 2004 (17%)
Process	*Publications & acceptance rate*
BGML design QoS evaluation CoSMIC tool suite Skoll integration	International Conference on Software Reuse (ICSR) 2004 (22%) RTAS 2005 (30%) International Conference on Software Engineering (ICSE) 2005 International Journal of Embedded systems (invited) IEEE Software Studia Infomatica Universalis Journal
Performance Evaluation	*Publications & acceptance rate*
Benchmark suite for evaluating CCM	RTAS 2004 (25%) Elsevier Real-time Systems Journal

Table 4: Publication Summary: (First and Second author contributions)

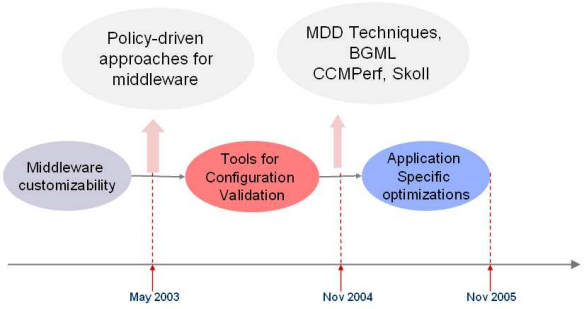

Figure 20: Research Time-line

OPTEML: Research Contributions	
Area	*Contribution*
General purpose optimization techniques	Patterns for fine-grain componentization of middleware architectures. Policy-driven middleware approaches
Configuration driven optimization techniques	DSMLs for capturing QoS evaluation concerns for component middleware & PLA
Partial specialization optimization techniques	FOCUS: A tool-driven approach for middleware specialization.
Preserving & validating middleware configuration properties	Model-driven DCQA process for preserving & validating middleware configuration properties. Main-effects screening a DCQA process for identifying configuration main-effects

Table 5: Summary of Research Contributions

resulted in several contributions to the development of CoSMIC and Skoll including: (1) Development of BGML [43] DSML, (2) Integration of BGML with CoSMIC and tool-driven approach for QoS evaluation [44] and (3) Development of Skoll main-effects screening process [96] and how a model-driven approach validates middleware configurations [45]. The final phase of OPTEML (proposed completion as of Nov 2005) will devise techniques for a tool driven process for middleware customization. Table 4 tabulates the conferences and their acceptance rates while Table 5 provides an overview of research contributions.

References

[1] S. Abramov and N. 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. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU, University of Copenhagen, Denmark, 1995.

[3] M. L. Bailey, B. Gopal, P. Sarkar, M. A. Pagels, and L. L. Peterson. Pathfinder: A pattern-based packet classifier. In *Proceedings of the 1st Symposium on Operating System Design and Implementation*. USENIX Association, Nov. 1994.

[4] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 190–199, San Francisco, CA, Mar. 2005.

[5] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[6] D. F. Box, D. C. Schmidt, and T. 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.

[7] D. Brunsch, C. O'Ryan, and D. 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.

[8] Center for Distributed Object Computing. The ZEN ORB. www.zen.uci.edu, University of California at Irvine.

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

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

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

[12] A. Corsaro, D. C. Schmidt, R. Klefstad, and C. O'Ryan. Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications. In *Proceedings of the 9^{th} Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, Sept. 2002.

[13] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

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

[15] M. Deshpande, D. C. Schmidt, C. O'Ryan, and D. 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.

[16] Z. D. Dittia, G. M. Parulkar, and J. 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, Apr. 1997. IEEE.

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

[18] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. 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, Oct. 1991. ACM.

[19] D. R. Engler and M. F. 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, Aug. 1996. ACM Press.

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

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

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

[23] C. Z. D. Gao and H.-A. Jacobseon. Towards just in time middleware architectures. In *Proceedings of the 2005 Aspect Oriented Software Engineering Conference (AOSD)*, Nov 2005.

[24] Gautam Thaker et. al. Implementation Experience with OMG's SCIOP Mapping. In *Proceedings of the 5^{th} International Symposium on Distributed Objects and Applications*, Nov. 2003.

[25] C. Gill, D. C. Schmidt, and R. Cytron. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), Jan. 2003.

[26] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. 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*, 2004.

[27] A. Gokhale and D. C. Schmidt. Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA. In *Proceedings of GLOBECOM '97*, Phoenix, AZ, Nov. 1997. IEEE.

[28] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. 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), Oct. 2002.

[29] 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, Sept. 2001.

[30] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, Oct. 1997. ACM.

[31] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. 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.

[32] J. Hu and D. C. Schmidt. JAWS: A Framework for High Performance Web Servers. In M. Fayad and R. Johnson, editors, *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. Wiley & Sons, New York, 1999.

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

[34] Institute for Software Integrated Systems. The ADAPTIVE Communication Environment (ACE). www.dre.vanderbilt.edu/ACE/, Vanderbilt University.

[35] V. Itkin. On partial and mixed program execution. In *Program Optimization and Transformation*, pages 17–30. Novosibirsk: Computing Center, 1983. (In Russian).

[36] M. Jayaram and R. 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, Feb. 1996.

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

[38] 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.

[39] R. Klefstad, A. S. Krishna, and D. C. Schmidt. Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, and Embedded CORBA Applications. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications*, Irvine, CA, October/November 2002. OMG.

[40] A. S. Krishna, D. C. Schmidt, and R. Klefstad. Enhancing Real-time CORBA via Real-time Java. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, Tokyo, Japan, May 2004. IEEE.

[41] A. S. Krishna, D. C. Schmidt, R. Klefstad, and A. Corsaro. Real-time CORBA Middleware. In Q. Mahmoud, editor, *Middleware for Communications*. Wiley and Sons, New York, 2003.

[42] A. S. Krishna, D. C. Schmidt, R. Klefstad, and A. 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.

[43] A. S. Krishna, D. C. Schmidt, A. Porter, A. Memon, and D. Sevilla-Ruiz. Improving the Quality of Performance-intensive Software via Model-integrated Distributed Continuous Quality Assurance. In *Proceedings of the 8th International Conference on Software Reuse*, Madrid, Spain, July 2004. ACM/IEEE.

[44] A. S. Krishna, E. Turkay, A. Gokhale, and D. C. Schmidt. Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 180–189, San Francisco, CA, Mar. 2005.

[45] A. S. Krishna, C. Yilmaz, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Preserving Distributed Systems Critical Properties: a Model-Driven Approach. *IEEE Software special issue on Persistent Software Attributes*, November/December 2004.

[46] A. S. Krishna, C. Yilmaz, A. Porter, A. Memon, D. C. Schmidt, and A. Gokhale. Distributed continuous quality assurance process for evaluating qos of performance intensive software. *Studia Infomatica Universalis*, 4, Mar. 2005.

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

[48] L. Lei, G.-H. Moll, and J. Kouloumdjian. A deductive database architecture based on partial evaluation. *SIGMOD Record*, 19(3):24–29, September 1990.

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

[50] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1997.

[51] 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, Apr. 2001.

[52] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Real-Time Systems Journal*, 23(1/2):85–126, July 2002.

[53] R. Marlet, S. Thibault, and C. Consel. Efficient implementations of software architectures via partial evaluation. *Journal of Automated Software Engineering*, 6(4):411–440, Oct. 1999.

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

[55] S. McCanne and V. 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, Jan. 1993.

[56] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

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

[58] J. C. Mogul, R. F. Rashid, and M. 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)*, Nov. 1987.

[59] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of the* 1^{st} *Symposium on Operating Systems Design and Implementation*. USENIX Association, Nov. 1994.

[60] 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.

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

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

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

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

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

[66] C. Pu, T. Autery, A. Black, C. Consel, C. Cowan, J. W. Jon Inouye, Lakshmi Kethana, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Symposium of Operating System Principles*, Copper Mountain Resort, Colorado, Dec. 1995.

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

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

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

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

[71] 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.

[72] M. Rutherford and A. 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, Sept. 2003.

[73] C. Sakama and H. Itoh. Partial evaluation of queries in deductive databases. *New Generation Computing*, 6(2,3):249–258, 1988.

[74] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan. Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes. In *Proceedings of the Winter USENIX Conference*, pages 85–106, San Diego, CA, Jan. 1993.

[75] R. E. Schantz and D. C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In J. Marciniak and G. Telecki, editors, *Encyclopedia of Software Engineering*. Wiley & Sons, New York, 2002.

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

[77] D. Schmidt and S. Vinoski. Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Request. *C++ Report*, 8(2), Feb. 1996.

[78] D. Schmidt and S. Vinoski. Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool. *C++ Report*, 8(4), Apr. 1996.

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

[80] D. C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching. In *Proceedings of the* 1^{st} *Annual Conference on the Pattern Languages of Programs*, pages 1–10, Monticello, Illinois, Aug. 1994.

[81] D. C. Schmidt, V. Kachroo, Y. Krishnamurthy, and F. Kuhns. Applying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications. *IEEE Communications Magazine*, 38(10):112–123, Oct. 2000.

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

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

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

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

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

[87] M. Sopitkamol and D. 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.

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

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

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

[91] 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.

[92] E. Turkay, A. Gokhale, and B. Natarajan. Addressing the Middleware Configuration Challenges using Model-based Techniques. In *Proceedings of the 42nd Annual Southeast Conference*, Huntsville, AL, Apr. 2004. ACM.

[93] T. Veldhuizen. Using c++ template metaprograms. *C++ Report*, 7(4):36–34, 1999.

[94] A. Vogel, B. Gray, and K. 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.

[95] C. F. J. Wu and M. Hamada. *Experiments: Planning, Analysis, and Parameter Design Optimization*. Wiley, 2000.

[96] C. Yilmaz, A. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, St. Louis, MO, May 2005.

[97] 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*, Jan. 1994.

[98] 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.