A Model-transformation Approach to Improving the Quality of Software Architectures for Distributed Real-time and Embedded Systems

Amogh Kavimandan[1] and Aniruddha Gokhale[1⋆]

ISIS, Dept. of EECS, Vanderbilt University, Nashville, TN, USA
{amoghk,gokhale}@dre.vanderbilt.edu

Abstract. The quality of a software architecture for component-based distributed systems is defined not just by its source code but also by other systemic artifacts, such as the assembly, deployment, and configuration of the application components and their component middleware. In the context of distributed, real-time, and embedded (DRE) component-based systems, bin packing algorithms and schedulability analysis have been used to make deployment and configuration decisions. However, these algorithms make only coarse-grained node and priority assignments but do not indicate how components are allocated to different middleware containers on the node, which are known to impact runtime system performance and resource consumption. This paper presents a model transformation-based algorithm that combines user-specified quality of service (QoS) requirements with the node assignments to provide a finer level of granularity and precision in the deployment and configuration decisions. A beneficial side effect of our work lies in how these decisions can be leveraged by additional backend performance optimization techniques. We evaluate our approach and compare it against the existing state-of-the-art in the context of a representative DRE system.
Keywords: Model-driven engineering, Graph/model transformations, component-based systems, deployment.

1 Introduction

Component-based software engineering (CBSE) has received much attention over the past few years to develop distributed systems including distributed, real-time, and embedded (DRE) systems, such as emergency response systems, aircraft navigation and command supervisory systems, and total shipboard computing systems. CBSE provides a simplified programming model and various mechanisms to separate functional and non-functional concerns of the system being designed, which lends it to rapid prototyping, (re-) configuration, and easier maintenance of DRE systems.

DRE systems have stringent runtime quality of service (QoS) requirements including predictable end-to-end latencies, reliability and security, among others. Naturally, the software architecture of the DRE system plays an important role in ensuring that the runtime QoS needs of DRE systems are met. In component-based DRE systems,

⋆ Contact Author

the software architecture is defined not just by the source code of the application functionality but also by a wide range of systemic issues including the assembly of application components, their deployment on the target nodes of the system and allocation of resources such as the CPU, and the component middleware that hosts the application components.

In component middleware, such as the CORBA Component Model (CCM) and Enterprise Java Beans (EJB), a container is a central concept that hosts application components. Containers hosting DRE system components provide a high degree of configurability by allowing (1) the choice of the number of thread resources to be configured for each component, their type (*i.e.*, static or dynamic), and their attributes, such as stack-size, etc., (2) control over asynchronous event communication, and event filtering and delivery options, and (3) control over client request invocation priorities on the server component. The *configuration space* – identified by all the mechanisms for specifying system QoS and their appropriate values – becomes highly complex. Thus, making the right configuration decisions is one key factor that determines the quality of the DRE system software architecture.

Prior research in improving the quality of DRE systems software architectures has focused on: (1) analysis-driven decomposition [1] of DRE system functionality into reusable application components that can be assembled and deployed; (2) component-to-node assignment [2] and resource allocations [3], and (3) schedulability and timing analysis [4, 5] to determine exact priorities and time periods for applications, which help in partitioning the system resources and configuring the middleware.

Despite these advances, key issues still remain unresolved in the deployment and configuration problem space of DRE system software architectures. For example, although bin packing algorithms [2] make effective decisions on component deployment, and schedulability analysis determines the priorities at which the components should execute, both these decisions are at best coarse-grained since they determine only the nodes on which the components must be deployed but do not indicate how they are deployed within the containers of the component middleware. The lack of finer-grained decisions often lead to suboptimal runtime QoS since these decisions are now left to application developers, who are domain experts but often lack detail understanding of the middleware.

To address these limitations, this paper presents a heuristics-based algorithm implemented within a model-transformation [6, 7] framework that combines models of user-specified QoS requirements, node assignment decisions, and priority values, and transforms them into optimal middleware configurations thereby enhancing the quality of the DRE system software architecture. Our research prototype has been implemented using the GReAT [8] model transformation framework for the Lightweight CCM (LwCCM) [9] middleware.

Two significant benefits accrue from our approach. First, by realizing the heuristic-based algorithm as an automated model transformation process, it can be seamlessly reapplied and reused during the iterative DRE system development process. Second, the configurations generated by our algorithm can be leveraged by additional backend optimization tools and techniques, such as the Physical Assembly Mapper (PAM) [10]

which reduces time and space overheads by merging collocated components at system deployment-time.

This paper is organized as follows: Section 2 discusses the challenges developers face in achieving optimal QoS configuration for DRE systems; Section 3 presents the overall approach taken, the enabling technologies used in our technique, and the model transformation algorithm we have developed; Section 4 empirically evaluates our approach in the context of a representative case study; Section 5 discusses the related work in the area; Section 6 gives concluding remarks.

2 Impediments to the Quality of DRE System Software Architectures

We now present the deployment and configuration-imposed impediments to the quality of DRE systems software architectures. We focus on issues that are both innate to the underlying middleware platforms as well as those that are accidental.

2.1 Overview of a Real-time Component Middleware

To better articulate the challenges we address in this paper, we first present an overview of a representative component middleware, which forms an integral part of a DRE system software architecture. Note, however, that our solution approach is general and not specific to the outlined middleware. Figure 1 illustrates the Lightweight CORBA Component Middleware (LwCCM) [9] architecture. DRE system developers can realize large-scale DRE systems by assembling and deploying LwCCM components. The applications within these DRE systems can use publish/subscribe communication semantics (by using the component event source and sink ports) or request/response communication semantics (by using the facet and receptacle ports).

In the context of component middleware platforms, such as LwCCM, a container is an execution environment provided for hosting the components such that they can access the capabilities of the hardware, networking and software (OS and middleware) resources. In particular, containers act as a higher-level of abstraction for hosting the components in which all the developer-specified QoS policies can be properly configured. Components with similar QoS configuration specifications are hosted within the same container so that all components in that container obtain the same QoS capabilities. Note that the bin packing algorithms described earlier cannot make these fine-grained decisions.

LwCCM – and in particular its container – leverages Real-time CORBA (RTCORBA) [11] to support the real-time QoS properties. RTCORBA in turn extends traditional CORBA artifacts, such as (a) the object request broker (ORB), which mediates the request handling between clients and servers, (b) the portable object adapter (POA), which manages the lifecycle of CORBA objects, (c) stubs and skeletons, which are generated by an interface definition language (IDL) compiler that hide the distribution aspects from the communicating entities, with real-time policies and interfaces.

RTCORBA (and hence LwCCM) defines standard interfaces and QoS policies that allow applications to configure and control (1) *processor resources* via thread pools,

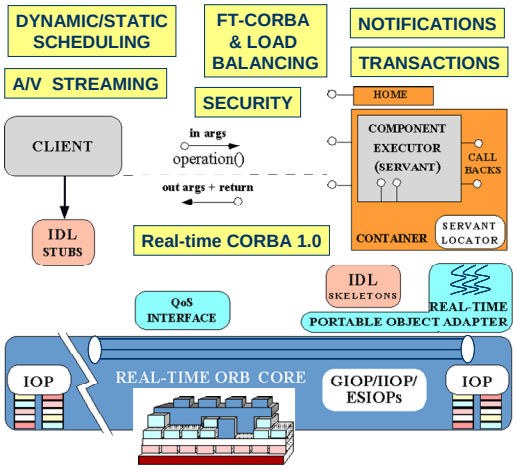

Fig. 1. Lightweight CORBA Component Model Architecture

priority mechanisms, intra-process mutexes, and a global scheduling service, (2) *communication resources* via protocol properties and explicit bindings, and (3) *memory resources* via buffering requests in queues and bounding the size of thread pools. For example, the priority at which requests must be handled can be propagated from the client to the server (the `CLIENT_PROPAGATED` model) or declared by the server (the `SERVER_DECLARED` model).

2.2 Inherent and Accidental Complexities in Deployment and Configuration

In the CBSE paradigm, application developers must determine how to deploy components within the containers, and grapple with the multiple different configuration options provided by the containers. In this context we outline two critical challenges impeding the quality of DRE system software architectures.

Challenge 1: Inherent Challenges in Deployment and Configuration. The different analyses techniques used in the development of DRE systems (*e.g.* schedulability and timing analysis) and deployment and resource allocation decisions (*e.g.*, where each component resides in the available computing node farm) dictate what QoS configurations are chosen for individual components of the application. For example, as shown earlier, LwCCM provides configuration mechanisms to assign priorities to every component, defines a fixed/variable priority request invocation and handling model (`PriorityModelPolicy`), allows defining the number of thread resources, their type (*i.e.*, static or dynamic), and concurrency options (`ThreadPool`).

For a component-based application, the mapping of the above analyses onto these available policies results in a number of unique QoS configurations, and naturally, as many containers. Unfortunately, the principles of separation of concerns in the design

of containers in component middleware architectures force service request invocations between components hosted on different containers to go through the typical request demultiplexing layers and marshaling/demarshaling and mechanisms even though they may be hosted in the same address space of the application server. Therefore, such invocations are considerably slower than the invocations between components that share the same container [12].

Thus, in effect, components placed on different containers (which are in turn created from unique QoS configurations) are unable to exploit the collocation optimizations performed by the middleware.[1] As such, the sub-optimal QoS configuration of the application leads to increased average end-to-end latencies. Since DRE systems are made up of hundreds of components, as the number of components in the system that are sub-optimally configured increases, the adverse impact on end-to-end latencies can significant.

Challenge 2: Accidental Complexities in Deployment and Configuration. It may be argued that the developers can keep track of the QoS configurations that are produced, and depending on DRE system QoS needs, make decisions on how to minimize the containers. Such a manual approach, however, introduces several non-trivial challenges for the application developers:

- Large-scale DRE systems typically consist of hundreds of components spanning multiple assemblies of components. Manually keeping track of all the configurations (and potentially combining them to minimize the number of containers) in such large-scale systems is very difficult and in some cases infeasible.
- Development of DRE systems is often an iterative process where new requirements are added. Thus, the system configuration needs to evolve accordingly to cater to new requirements, and the optimizations listed above need to be performed at the end of each reconfiguration cycle.
- The configuration optimization activity forces the developers to have a detailed knowledge of the middleware platform. Further, the activity itself is not central to the development of application logic and may in fact be counter-productive to the promise of CBSE.

Addressing both these challenges calls for automated tools and techniques to perform the deployment and configuration optimizations so that the quality of the resulting DRE system software architecture is enhanced.

3 Enhancing the Quality of DRE System Software Architectures

We now present our model transformation-based approach to address the impediments to the quality of software architectures of component-based DRE systems stemming from suboptimal deployment and configuration decisions. We use a simple representative example to discuss our approach.

[1] Many middleware optimize the communication path for entities that reside in the same address space.

3.1 Representative Case Study

The Basic Single Processor (BasicSP) scenario shown in Figure 2 is a reusable component assembly available in the Boeing Bold Stroke [13] component avionics computing product line. BasicSP uses a publish/subscribe service for event-based communication among its components, and has been developed using a component middleware platform, such as LwCCM.

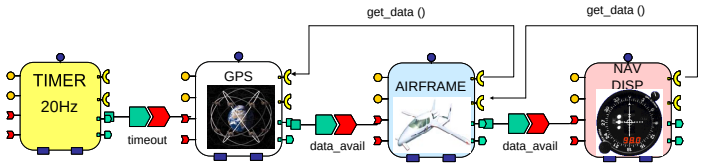

Fig. 2. Basic Single Processor Component Assembly

A GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are at 20 Hz. The scenario shown in Figure 2 begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer*, the *GPS* component generates its data and issues a data available event. The *Airframe* component retrieves the data from the *GPS* component, updates its state, and issues a data available event. Finally, the *NavDisplay* component retrieves the data from the *Airframe* and updates its state and displays it to the pilot.

In its normal mode of operation, the *Timer* component generates pulse events at a fixed priority level, although its real-time configuration can be easily changed such that it can potentially support multiple priority levels.

It is necessary to carefully examine the end-to-end application critical path and configure the system components correctly such that the display refresh rate of 20 Hz may be satisfied. In particular, the latency between *Timer* and *NavDisplay* components needs to be minimized to achieve the desired end goal. To this end, several characteristics of the BasicSP components are important and must be taken into account in determining the most appropriate QoS configuration space.

For example, the *NavDisplay* component receives update events only from the *Airframe* component and does not send messages back to the sender, *i.e.*, it just plays the role of a client. The *Airframe* component on the other hand communicates with both the *GPS* and *NavDisplay* components thereby playing the role of a client as well as a server. Various QoS options provided by the target middleware platform (in case of BasicSP, it is LwCCM) must ensure that these application-level QoS requirements are satisfied. For achieving the goal of reducing the latency between *Timer* and *NavDisplay* components, it is crucial to carefully analyze the QoS options chosen for each component in BasicSP, and exploit opportunities to either reuse or combine them such that this goal can be met.

The system is deployed on two physical nodes. Application developers of our case study choose a modeling environment to model the BasicSP component assembly and annotate its real-time requirements as shown in Figure 3. We have used the Generic Modeling Environment (GME) [14] for modeling the DRE system. GME provides a

graphical user interface that can be used to define both modeling language semantics and system models that conform to the languages defined in it. *Model interpreters* can be developed using the generative capabilities in GME that parse and can be used to generate deployment, and configuration artifacts for the modeled application.

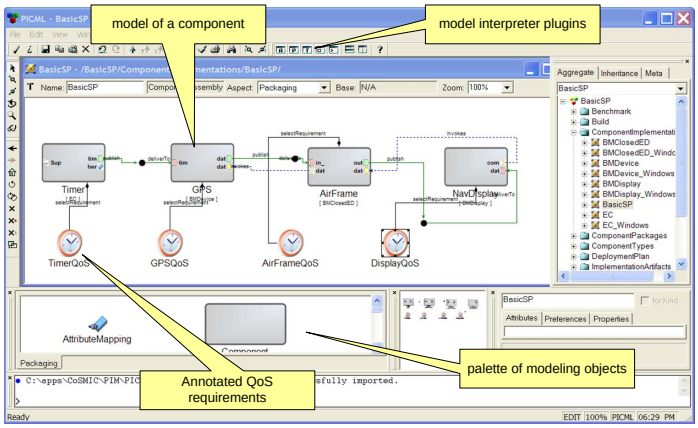

Fig. 3. BasicSP Model

The developers use the model interpreter plugins shown in the figure to automatically synthesize deployment and configuration metadata that describes how the components are assigned to nodes of the operating environment, and what configuration options of the component middleware are to be used for each component. These model interpreters encapsulate the bin packing and schedulability analyses algorithms alluded to earlier.

The generated metadata is usually in the form of XML, which is then parsed by the underlying middleware's deployment and configuration tool to deploy and configure the DRE system before operationalizing it. As mentioned earlier, due to a lack of finer-grained decisions, the generated XML metadata will often result in DRE system software architectures that perform suboptimally.

3.2 Heuristics-based Model-transformation Algorithm

The model transformation algorithm we developed takes the following models and generated artifacts as its input: (1) DRE QoS requirements specification in the form of models as shown in Figure 3, and (2) the generated DRE system deployment plan indicating the coarse-grained component-to-node mapping and configuration options to be used for the middleware. We assume that this mapping includes collocation groups, which are sets that include the components that can be placed together on a node and that too in the same address space. The objective of our algorithm is to improve the end-to-end latencies in DRE system as well as reduce the memory footprint of the DRE system by virtue of minimizing the number of containers needed to host the DRE system components.

The output of our algorithm is an enhanced QoS policy set[2], which is incorporated into the DRE system model. Our approach produces optimized QoS policy sets by employing novel ways of reusing and/or combining existing deployment and configuration metadata and applying deployment heuristics in an application-specific manner.

We have used the Graph Rewriting And Transformation (GReAT) [8] language for defining our transformation algorithms. GReAT, which is developed using GME, can be used to define transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution engine (GR-Engine). The graph rewriting rules are defined in GReAT in terms of source and target languages (*i.e.*, metamodels).

Below we explain the individual steps in our transformation process.

Step I: Modeling Language used in the Transformation Algorithm To demonstrate our technique we required a modeling language to enable the developers to annotate their QoS requirements on the DRE system models. A simplified UML QoS configuration metamodel that we used is shown in Figure 4.

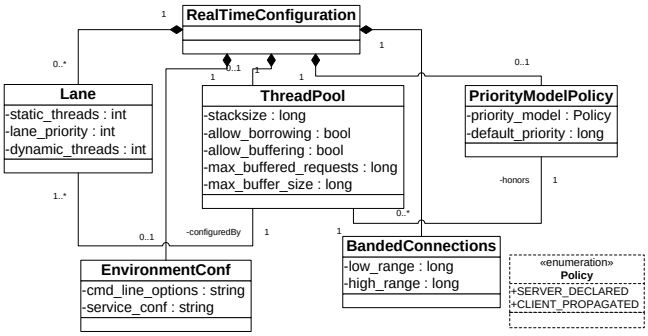

Fig. 4. Simplified UML notation of QoS configuration meta-model in CQML

The metamodel defines the following elements corresponding to several LwCCM real-time configuration mechanisms:

- `Lane`, which is a logical set of threads each one of which runs at `lane_priority` priority level. It is possible to configure the number of *static* threads (*i.e.*, those that remain active till the system is running, and *dynamic* threads (*i.e.*, those threads that are created and destroyed as required) using `Lane` element.
- `ThreadPool`, which controls various settings of `Lane` elements, or a group thereof. These settings include `stacksize` of threads, whether borrowing of threads across two `Lane` elements is allowed, and maximum resources assigned to the buffer requests that cannot be immediately serviced.

[2] QoS policy set is a group of configuration files that completely capture the DRE system QoS. These files are used by the middleware to ultimately provision infrastructure resources such that the QoS can be met.

- `PriorityModelPolicy`, which controls the policy model that a particular `ThreadPool` follows. It can be set to either CLIENT_PROPAGATED if the invocation priority is preserved, or SERVER_DECLARED if the server component changes the priority of invocation.
- `BandedConnections`, which defines separate connections for individual (client) service invocations. Thus, using `BandedConnections`, it is possible to define a separate connection for each (range of) service invocation priorities of a client component. The range can be defined using `low_range` and `high_range` option values of *BandedConnections*.

Step II: QoS Policy Optimization Algorithm Algorithm 1 depicts our heuristics-based model-transformation algorithm, which uses the metamodel shown in Figure 4 as its source and target language, for optimizing the deployment and configuration decisions.

Algorithm 1: Transformation Algorithm for Optimizing Deployment and Configuration Metadata

Input: set of deployment plans SP;
set of components SC, SCS (those that use server declared policy), SCC (those that use client propagated policy);
component c, c_p;
deployment plan p;
set of QoS policies SQ_1, SQ_2, qp_a (QoS policy set of a specific component 'a'), qp_b (similarly for a component 'b');
set of collocation groups SCG;
collocation group g

1 **begin**
2 **foreach** $p \in SP$ **do**
3 $SCG \leftarrow collocationGroups(p)$;// collect all collocation groups in the deployment plan
4 **foreach** $g \in SCG$ **do**
5 $SC \leftarrow SC + components(g)$; // Collect all components of a single collocation group
6 **if** $c \in SC \mid c.priorityModel == SERVER\_DECLARED$ **then**
7 $SCS \leftarrow SCS + c$; // Collect all components using the server declared policy
8 **else if** $c \in SC \mid c.priorityModel == CLIENT\_PROPAGATED$ **then**
9 $SCC \leftarrow SCC + c$; // Collect all components using the client propagated policy
10 **foreach** $c \in SCS$ **do**
11 $SQ_1 \leftarrow SQ_1 + c.QoSPolicy()$;
12 minimize SQ_1
13 subject to $qp_a \bowtie qp_b \mid qp_a \cong qp_b$;
14 **end**
15 **foreach** $c \in SCC$ **do**
16 $SQ_2 \leftarrow SQ_2 + c.QoSPolicy()$;
17 minimize SQ_2
18 subject to $qp_a \bowtie qp_b \mid qp_a \cong qp_b$;
19 **end**
20 **end**
21 $modifyDeploymentPlan(p, SQ_1, SQ_2)$; // modify the plan and repeat until no more optimizations are feasible
22 **end**
23 **end**

The algorithm is executed for all the deployment plans specified for an application and the policy optimizations are applied for each such plan as shown in Line 2. In Line 5, all the components from a single collocation group are found.[3] Based on whether they have SERVER_DECLARED or CLIENT_PROPAGATED priority model, they are grouped together in *SCS* and *SCC* as shown in Lines 7 and 9, respectively.

[3] Note that this is a host-based collocation group.

Finally, for each set of components above, the algorithm minimizes the number of QoS policies in Line 12 subject to the condition in Line 13. This condition stipulates that if QoS policies of two (sets of) components a and b each indicated in the Algorithm by qp_a and qp_b, respectively, are similar (binary Boolean function $\cong$ finds whether the policies are similar), then they are combined (indicated by $\bowtie$) leading to a reduction in the size of SQ_1. This test is applied pairwise to all components in the set. The Algorithm implements symmetric rules for CLIENT_PROPAGATED policy model.

In Line 21 the results from applying all the above rules to the DRE system model are used to modify the current deployment plan, and the process is repeated for all the remaining plans of the DRE system until no more optimizations are feasible.

4 Evaluating the Merits of the Transformation Algorithm

This section evaluates our approach to optimizing the original deployment and configurations for component-based DRE systems. We claim that the quality of the resulting software architecture is improved if it is able to demonstrate an improved performance. We describe our results in the context of our case study explained in Section 3.1. We show how the end-to-end latency results after applying our algorithm achieves considerable improvement over the existing state-of-the-art. Moreover, we also demonstrate a beneficial side effect of our solution by discussing the algorithm can be combined with additional backend optimization frameworks like the Physical Assembly Mapper (PAM) [10].

4.1 Experimental Setup & Empirical Results

We have used ISISLab (`www.dre.vanderbilt.edu/ISISlab`) for evaluating our approach. Each of the physical nodes used in our experiments was a 2.8 GHz Intel Xeon dual processor, 1 GB physical memory, 1 GHz network interface, and 40GB hard disks. Version 0.6 of our Real-time LwCCM middleware called CIAO was used running on Redhat Fedora Core release 4 with real-time preemption patches. The processes that hosted BasicSP components were run in the POSIX scheduling class SCHED_FIFO, enabling first-in-first-out scheduling semantics based on the priority of the process.

To showcase our results, we first modeled the the BasicSP scenario and generated the deployment and configuration metadata for each of its components. Note that the metadata is generated using the model interpreters that encapsulate appropriate bin packing and schedulability analysis techniques. We collected the end-to-end latency metrics for the BasicSP scenario using the initial deployment and configuration metadata.

We then applied the transformation algorithm 1 to our BasicSP model which resulted in more fine-grained optimizations to the existing deployment and configuration metadata. The BasicSP scenario was then executed again with the updated QoS policies, and the results were collected. For both these experiments, the results were obtained by repeating invocations for 100,000 iterations after 10,000 warmup iterations and averaging them.

Figures 5 and 6 show the results of applying our approach to BasicSP scenario comparing them to those derived from the original deployment and configurations. The figure plots the average end-to-end latency and its standard deviation for the invocations from *Timer* to *NavDisplay* components in BasicSP with and without our approach.

As shown in Figure 5, the average latency improved by ∼70% when our technique was used for optimizing BasicSP QoS configurations. The standard deviation on the other hand, improved by ∼59% as plotted in Figure 6.

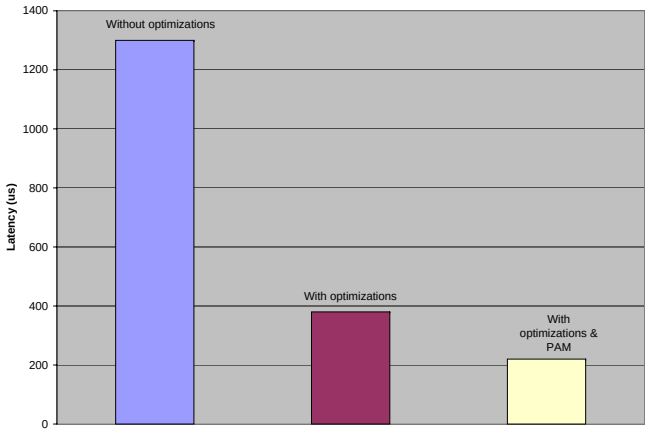

Fig. 5. Average end-to-end Latency

Without our approach, the initial BasicSP QoS configuration contained separate policies for each of its four components. Out of the four components, only the *Timer* component has SERVER_DECLARED priority model, while the rest of the components have CLIENT_PROPAGATED priority model. Thus, as indicated on Lines 12 and 13, when Algorithm 1 is applied to BasicSP, the QoS policy set is reduced to a size of two, one for each kind of priority model. This reduction in the size of the QoS policy set leads to the ∼70% improvement in end-to-end latency between *Timer* and *NavDisplay* components.

The third graph in each figure indicates the additional improvements in end-to-end latencies accrued as a result of leveraging backend optimization frameworks, such as PAM [10]. PAM is a deployment-time technique that *fuses* a set of components collocated in a container to reduce memory footprint and latency between service invocations. Our approach simply indicates what components should be part of the same container, however, individual components continue to require their own stubs and skeletons, and other glue code, which continues to be a source of memory footprint overhead. Approaches like PAM can then be used to eliminate this remaining overhead.

As shown in Figures 5 and 6, when applied in conjunction with PAM, our approach leads to a combined improvement of ∼83% in the end-to-end latency and ∼65% in the observed standard deviation in latency for BasicSP scenario.

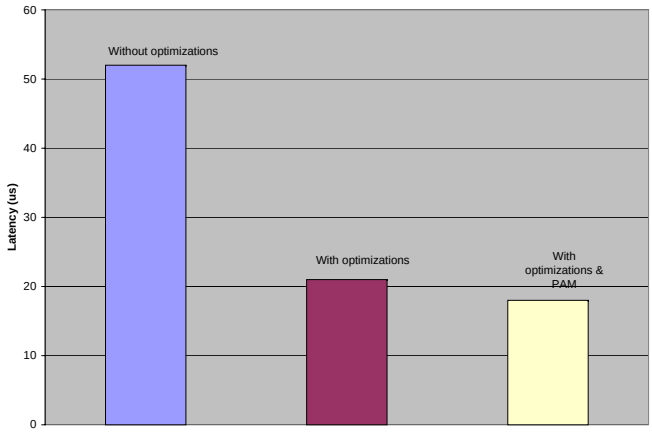

Fig. 6. Standard Deviation in Latency

4.2 Discussion

Our transformation algorithm described in Sections 3.2 relies on QoS configuration analyses in a platform-specific manner. We specifically showed how it has been realized in the context of a LwCCM middleware implementation. Naturally to extend it to other middleware platforms requires a careful study of the other platform's configuration space.

The results indicated an improvement of $\sim$70% in invocation latency between an execution path consisting of four components (the execution path here refers to the invocations from *Timer*, to *GPS*, to *AirFrame*, and finally to *NavDisplay* components in BasicSP). Recall that the BasicSP is an assembly of components that is used in the context of larger DRE system architectures. With increasing scale of the DRE system, it becomes necessary to leverage every opportunity for optimizations.

We expect the improvements accrued using our approach to be even higher. This is because the reduction in end-to-end latency is dependent upon how effectively the QoS policy sets SQ_1 and SQ_2 in Algorithm 1 are minimized. Large-scale DRE systems would have a number of QoS policies specified across their component assemblies, and in general, would be expected to have more opportunities to combine and reuse these policies leading to further latency improvements.

5 Related Work

Since the work presented in this paper results in performance optimizations to the underlying middleware thereby improving the quality of the DRE system software architecture, we compare our work with synergistic works. Moreover, since our research is applicable at design-time, we focus primarily on design- and deployment-time techniques to compare our work against.

Design-time approaches to component middleware optimization include eliminating the dynamic loading of component implementation shared libraries and establishing

connections between components done at runtime, as described in the static configuration of CIAO [15]. Our approach is different since it uses model transformations of configurations at design-time. Our approach is thus not restricted to optimizing just the inter-connections between components. Moreover, the static configuration approach can be applied in combination to our approach.

Another approach to optimizing the middleware at design/development-time employs context-specific middleware specializations for product-line architectures [16]. This work is based on utilizing application-, middleware- and platform-level properties that do not vary during the normal application execution in order to reduce the excessive overhead caused by the generality of middleware platforms.

Some work has also been done in the area of Aspect-Oriented Programming (AOP) techniques that rely essentially on automatically deriving subsets of middleware based on the use-case requirements [17], and modifying applications to bypass middleware layers using aspect-oriented extensions to CORBA Interface Definition Language (IDL) [18]. In addition, middleware has been synthesized in a "just-in-time" fashion by integrating source code analysis, and inferring features and synthesizing implementations [19].

Contrary to the above approaches, our model transformation-based technique relies only on the specified (1) QoS requirements specification and (2) the initial deployment plan, in order to optimize the QoS policies. Our approach does not necessitate any modifications to the application, *i.e.*, the application developer need not design his/her application tuned for a specific deployment scenario. As our results in Section 4.2 have indicated, our approach can be used in a complementary fashion to any of the design/development-time approaches discussed above, since there exist several opportunities for QoS optimization at various stages in application development.

Deployment-time optimizations research includes BluePencil [20], which is a framework for deployment-time optimization of web services. BluePencil focuses on optimizing the client-server binding selection using a set of rules stored in a policy repository and rewriting the application code to use the optimized binding. While conceptually similar, our approach differs from BluePencil because it uses models of application structure and application deployment to serve as the basis for the optimization infrastructure.

BluePencil uses techniques such as *configuration discovery* that extract deployment information from configuration files present in individual component packages. By operating at the level of individual client-server combinations, the QoS optimizations achieved in our transformation-based approach are non-trivial to perform in BluePencil. BluePencil also relies on modification to the application source code to rewrite the application code, while our approach is non-intrusive and does not require application source code modifications, and it only relies on the specified application policies and deployment plans.

Research on approaches to optimizing middleware at runtime has focused on choosing optimal component implementations from a set of available alternatives based on the current execution context [21]. QuO [22] is a dynamic QoS framework that allows dynamic adaptation of desired behavior specified in *contracts*, selected using proxy objects called *delegates* with inputs from runtime monitoring of resources by *system*

condition objects. QuO has been integrated into component middleware technologies, such as LwCCM.

Other aspects of runtime optimization of middleware include domain-specific middleware scheduling optimizations for DRE systems [23], using feedback control theory to affect server resource allocation in Internet servers [24] as well as to perform real-time scheduling in Real-time CORBA middleware [25]. Our work is targeted at optimizing the middleware resources required to host composition of components in the presence of a large number of components, whereas, the main focus of these related efforts is to either build the middleware to satisfy certain performance guarantees, or effect adaptations via the middleware depending upon changing conditions at runtime.

6 Concluding Remarks

The last few years have seen a significant increase in the popularity of component middleware platforms for developing distributed, real-time and embedded (DRE) systems, such as emergency response systems, intelligent transportation systems, total shipboard computing environment across a wide range of application domains. Its higher levels of programming abstractions coupled with mechanisms that support sophisticated and highly tunable infrastructure configuration are well suited for rapid development and/or maintenance of such systems.

The generality of contemporary component middleware platforms, however, has increased the complexity in properly configuring these platforms to meet application-level QoS requirements. Automated solutions [26] are an attractive alternative to achieving the QoS configuration of component-based systems, however, they incur excessive system resource overheads often leading to sub-optimal system QoS.

In this paper, we discussed an automated, model transformation-based approach that takes into account the component collocation heuristics to optimize application QoS configuration thereby improving the quality of the software architecture. We discussed the design of our approach, and the transformation algorithm used to optimize the QoS configuration. We also evaluated our approach and compare it against the existing state-of-the-art. The results demonstrated the effectiveness of our approach in optimizing QoS configuration in the context of a representative DRE system reusable component assembly.

The following are the lessons learned from our research:

- Optimal QoS configuration for component-based systems is a crucial research area that has been unaddressed till date. As component middleware gains popularity, and available resources become constrained, especially in the context of DRE systems, it is critical to improve the overall quality of the DRE system software architectures.
- Existing research works in QoS configuration have focused largely on achieving *locally optimized* solutions, *i.e.*, they are restricted to analyzing and modifying/manipulating the middleware configuration space.
- Our approach showed that excessive overheads can be avoided by analyzing QoS configurations in the context of other application characteristics such as its component collocation/node placement heuristics.

- We have focused on combining the deployment decisions with the application QoS specification. However, as our results have indicated, further optimizations are possible by combining our technique with other design-, development-, run-time techniques, which merits further investigation. Additional investigations are also necessary to test our approach on larger DRE systems and different middleware platforms.
- Our approach indicated improvements in latencies. Significant research remains to done to see how other QoS metrics can be improved as well. When multiple QoS metrics are considered together, simple heuristics may not work. Instead multi-objective optimizations [27] may be necessary. This will form part of our future research.

References

1. Hatcliff, J., Deng, W., Dwyer, M., Jung, G., Prasad, V.: 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) 160–172
2. de Niz, D., Rajkumar, R.: Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. International Journal of Embedded Systems **2**(3) (2006) 196–208
3. Ghosh, S., Rajkumar, R., Hansen, J., Lehoczky, J.: Integrated QoS-aware Resource Management and Scheduling with Multi-resource Constraints. Real-Time Syst. **33**(1-3) (2006) 7–46
4. Stankovic, J.A., Zhu, R., Poornalingam, R., Lu, C., Yu, Z., Humphrey, M., Ellis, B.: VEST: An Aspect-Based Composition Tool for Real-Time Systems. In: RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Washington, DC, USA, IEEE Computer Society (2003) 58
5. Gu, Z., Kodase, S., Wang, S., Shin, K.G.: A model-based approach to system-level dependency and real-time analysis of embedded software. In: RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Washington, DC, USA, IEEE Computer Society (2003) 78
6. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific Publishing Company (jan 1997)
7. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software **20**(5) (2003) 42–45
8. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the Use of Graph Transformation in the Formal Specification of Model Interpreters. Journal of Universal Computer Science **9**(11) (2003) 1296–1321 `www.jucs.org/jucs_9_11/on_the_use_of`.
9. Object Management Group: Lightweight CCM RFP. realtime/02-11-27 edn. (November 2002)
10. Balasubramanian, K., Schmidt, D.C.: Physical Assembly Mapper: A Model-driven Optimization Tool for QoS-enabled Component Middleware. In: Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications Symposium, St. Louis, MO, USA (April 2008) 123–134
11. Object Management Group: Real-time CORBA Specification. 1.2 edn. (January 2005)
12. Wang, N., Schmidt, D.C., Parameswaran, K., Kircher, M.: Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In: 24th Computer Software and Applications Conference, Taipei, Taiwan, IEEE (October 2000)

13. Sharp, D.C.: Reducing Avionics Software Cost Through Component Based Product Line Development. In: Software Product Lines: Experience and Research Directions. Volume 576. (Aug 2000) 353–370
14. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer **34**(11) (2001) 44–51
15. Subramonian, V., Shen, L.J., Gill, C., Wang, N.: The design and performance of configurable component middleware for distributed real-time and embedded systems. In: RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04), Washington, DC, USA, IEEE Computer Society (2004) 252–261
16. Krishna, A., Gokhale, A., Schmidt, D.C., Hatcliff, J., Ranganath, V.: Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures. In: Proceedings of EuroSys 2006, Leuven, Belgium (April 2006) 205–218
17. Hunleth, F., Cytron, R.K.: Footprint and Feature Management Using Aspect-oriented Programming Techniques. In: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 02), ACM Press (2002) 38–45
18. Ömer Erdem Demir, Dévanbu, P., Wohlstadter, E., Tai, S.: An Aspect-oriented Approach to Bypassing Middleware Layers. In: AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2007) 25–35
19. Zhang, C., Gao, D., Jacobsen, H.A.: Towards Just-in-time Middleware Architectures. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2005) 63–74
20. Lee, S., Lee, K.W., Ryu, K.D., Choi, J.D., Verma, D.: Ise01-4: Deployment time performance optimization of internet services. Global Telecommunications Conference, 2006. GLOBECOM'06. IEEE (Nov 2006) 1–6
21. Diaconescu, A., Mos, A., Murphy, J.: Automatic performance management in component based software systems. In: ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04), Washington, DC, USA, IEEE Computer Society (2004) 214–221
22. Zinky, J.A., Bakken, D.E., Schantz, R.: Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems **3**(1) (1997) 1–20
23. Gill, C.D., Cytron, R., Schmidt, D.C.: Middleware Scheduling Optimization Techniques for Distributed Real-time and Embedded Systems. In: Proceedings of the 7^{th} Workshop on Object-oriented Real-time Dependable Systems, San Diego, CA, IEEE (January 2002)
24. Zhang, R., Lu, C., Abdelzaher, T.F., Stankovic, J.A.: ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In: ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS), Washington, DC, USA (2002) 301
25. Lu, C., Stankovic, J.A., Son, S.H., Tao, G.: Feedback control real-time scheduling: Framework, modeling, and algorithms. Real-Time Syst. **23**(1-2) (2002) 85–126
26. Kavimandan, A., Gokhale, A.: Automated Middleware QoS Configuration Techniques using Model Transformations. In: Proceedings of the 14^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008), St. Louis, MO, USA (April 2008) 93–102
27. Deb, K., Gupta, H.: Searching for Robust Pareto-Optimal Solutions in Multi-objective Optimization. In: Proceedings of the Third International Conference on Evolutionary Multi-Criterion Optimization (EMO 2005). Lecture Notes in Computer Science, vol. 3410, Springer (March 2005) 150–164