Physical Assembly Mapper:
A Model-driven Optimization Tool for QoS-enabled Component Middleware*

Krishnakumar Balasubramanian, Douglas C. Schmidt
Dept. of EECS, Vanderbilt University, Nashville
{kitty,schmidt}@dre.vanderbilt.edu

Abstract

This paper provides four contributions to the study of optimization techniques for component-based distributed real-time and embedded (DRE) systems. First, we describe key challenges of designing component-based DRE systems and identify key sources of overhead in a typical component-based DRE system from the domain of shipboard computing. Second, we describe a class of optimization techniques applicable to the deployment of component-based DRE systems. Third, we describe the Physical Assembly Mapper (PAM), which is a model-driven optimization tool that implements these techniques to reduce footprint. Fourth, we evaluate the benefits of these optimization techniques empirically and analyze the results. Our results indicate that the deployment-time optimization techniques in PAM provides significant benefits, such as 45% improvement in footprint, when compared to conventional component middleware technologies.

1. Introduction
1.1. Overview and Challenges of Component Middleware for DRE Systems

Component middleware technologies have raised the level of abstraction used to develop DRE systems, such as avionics mission computing [1] and shipboard computing systems [2]. Example component middleware technologies include the Lightweight CORBA Component Model (CCM), Boeing's PRiSM, OpenCOM, nesC's component model, and Timing Definition Language extension to Giotto. Component middleware also promotes the decomposition of monolithic systems into collections of inter-connected components (called a *component assembly*) that is composed of individual components (called *monolithic components*).

Although component middleware provides many benefits, certain challenges have restricted its use for a large class of DRE systems. For example, while functional decomposition of DRE systems into component assemblies and

monolithic components helps promote reuse across product lines [3], it can also increase the number of components in the system, thereby significantly increasing the memory footprint of component-based DRE systems. Although many enterprise systems can offset the increase in footprint via hardware upgrades, DRE systems often have stringent footprint limitations due to size, weight, and power constraints.

Optimizing components manually in a large-scale DRE system is hard due to the sheer number of components; the usage of any single component also tends to span multiple compositional hierarchies, *e.g.*, a single component could be connected to different sets of components in different assemblies. Since components are often reused across a product line, an optimization applicable in one context may not be applicable in another. Optimizations should therefore be performed based on the requirements of every unique deployment.

1.2. Solution Approach → Physical Assembly Optimizer

To address the challenges of large-scale component-based DRE systems described above, we have developed model-driven optimization techniques that help reduce the space overhead in DRE systems. The optimization techniques described in this paper focus on reducing the footprint overhead in component middleware by optimizing the assembly of components at deployment-time, as opposed to design-time and/-or run-time. Our optimizations reduce the number of components required to deploy a DRE system using a technique known as *fusion*, which combines multiple components to create *physical assemblies*, as described below.

Assemblies of components defined by standard component middleware, such as Lightweight CCM, are typically *virtual*, *i.e.*, the individual components that form the assembly can be spread across multiple machines of the target domain. Figure 1a shows an example of a component assembly. In contrast to a *virtual* assembly, a *physical* assembly is defined as the set of components created from the monolithic components that are deployed onto a single process (a single address space) within a physical node. As shown in Figure 1b, a physical assembly is itself a full-fledged component, *i.e.*, it has a component interface and an implemen-

(a) Operational String (Pre-Optimization) (b) Optimized Operational String (Post-Optimization)
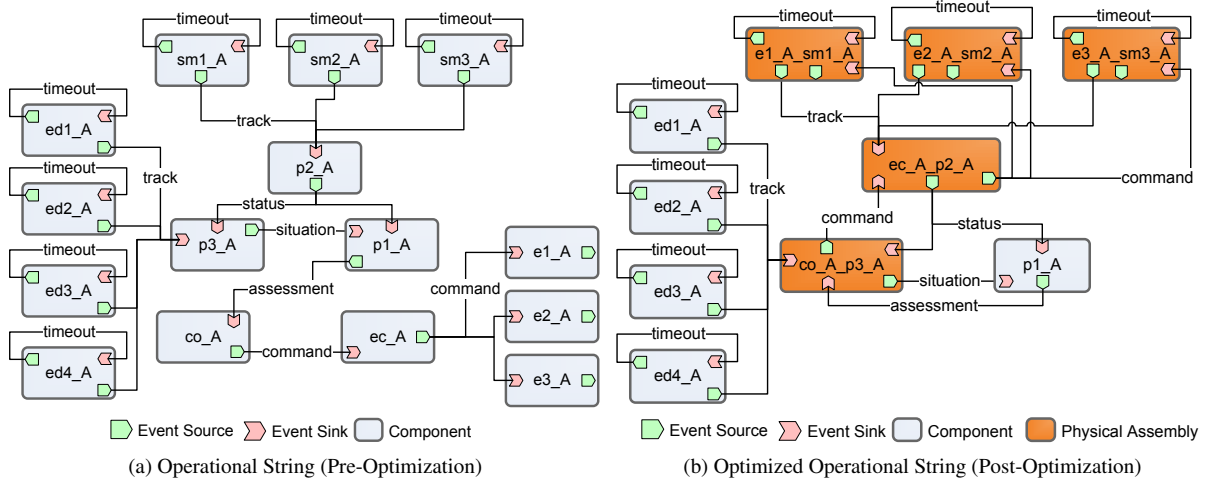
Figure 1: Shipboard Computing Application

tation. The implementation of the physical assembly, however, simply delegates to the original implementations of the monolithic components from which the physical assembly is created.

To create physical assemblies we require information on (1) *application structure*, *i.e.*, connections between components, (2) *application QoS configuration*, *i.e.* QoS configuration associated with each component, and (3) *application deployment*, *i.e.*, the mapping of components onto physical nodes(and processes within nodes). The optimization techniques described and evaluated in this paper obtain this information from models of the application built using domain-specific modeling languages (DSML)s. A DSML defines a type system that formalizes the application structure, behavior, and requirements within a particular domain, such as software defined radios, avionics mission computing, online financial services, warehouse management, or as in our case even the domain of component middleware itself.

We have applied these optimization techniques in a model-driven tool called the *Physical Assembly Mapper* (PAM) that optimizes component-based DRE systems developed using Lightweight CCM. PAM is built using the Generic Modeling Environment (GME) [4], which is a meta-programmable environment for creating DSMLs. PAM creates physical assemblies using a model of the connectivity information between components and the QoS policies. By operating at the high-level abstraction of models, PAM optimizes component assemblies across two dimensions—*footprint* and *performance*—and at multiple levels—*local* (deployment plan-specific) and *global* (application-wide). Since PAM's optimizer operates at deployment-time no changes are required to the mono-

lithic component implementations, functional decomposition, or structure of component-based DRE systems.

The novelty of PAM stems from identifying and applying component assembly optimizations automatically from models of applications. This model-driven approach eliminates the difficulties associated with applying these optimizations manually. Moreover, these optimizations cannot be performed in a generalized manner at the middleware level due to their context-dependent nature.

2. Challenges in Large-scale Component-based DRE systems

This section describes the application of component middleware programming models to a representative application from the shipboard computing domain. Using this application as an exemplar, we describe the cost of these features with respect to memory footprint for DRE systems. To make our discussion concrete, we use the Lightweight CCM as an example of component model for our discussion. The sources of overhead, however, are generally applicable to any layered component middleware, such as Enterprise Java Beans (EJB), Boeing's PRiSM, and OpenCOM.

2.1. Overview of the Shipboard Application

To evaluate the overhead of applying component middleware to the development of large-scale DRE systems, we use a shipboard computing application that runs in a metropolitan area network of computational resources and sensors to provide on-demand situational awareness and actuation capabilities for human operators and respond flexibly to unanticipated run-time conditions. To meet such demands in a robust and timely manner, the shipboard computing environment uses component-based services to bridge the gap between shipboard applications and the un-

derlying OS and middleware infrastructure to support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization.

The shipboard computing application we used for our experiments was developed using the CIAO middleware [2]. This application consists of a number of components grouped together into multiple *operational strings*. As shown in Figure 1a, an operational string is composed of a sequence of components connected together using the component's named ports. The connection labels (*e.g.*, `timeout,` `track` *et al*) in Figure 1a denote the name of the ports at the opposite ends of the connection. Component `ed1_A` thus has an event source port called `track` that sends events to an event sink port called `track` on component `p3_A`. Likewise, each component has an associated interface that represents the name of the component without the number, *e.g.*, component instance `ed1_A` implements the component interface type `ed_A`. The operational string in Figure 1a consists of four event detectors (sensors) (`ed1_A,` `ed2_A,` `ed3_A,` `ed4_A`) that gather data from the physical devices in a periodic fashion triggered by sensor-specific timeouts. There are also three system monitors (`sm1_A,` `sm2_A,` `sm3_A`) that monitor the overall system state and are triggered similar to the sensors. Both sensors and system monitors publish the data to a series of planners (`p3_A,` `p2_A,` `p1_A`). After analyzing the sensor data and the inputs from system monitors, the planners perform control decisions using a co-ordinator (`co_A`), an effector controller (`ec_A`) and three effectors (`e1_A,` `e2_A,` `e3_A`).

Each operational string contains up to 15 components, and the application used in our experiments is made up of 10 such operational strings, for a total of 150 components. Operational strings run at different importance levels, where the higher importance operational strings receive priority when accessing a resource. The application itself is deployed using 10 different deployment plans across 5 different physical nodes (bathleth, scimitar, rapier, cutlass, and saber). The assignment of components to nodes was determined *a priori* using high-level resource planning algorithms [5], and was available as input to our algorithms. Each node had a variable number of components, ranging from 20 to as high as 80 components assigned to it. Although the shipboard application is composed of 10 operational strings, to simplify the exposition we will explain our techniques throughout the paper using the single operational string shown in Figure 1a, whose components were all mapped onto a single node.

The contribution to the memory footprint of a component DRE system can be classified into two categories: *static* and *dynamic*. Static footprint increases result from code generated to integrate the implementation of a component with the middleware's run-time environment; such code generation is specific to each unique component type in the system. Dynamic footprint increases are due to the creation of run-time infrastructural elements, such as component homes and component context on a per-component basis. We discuss both types of memory footprint overhead below.

2.2. Static Footprint Overhead

As shown in Figure 2, for every component type in a DRE system, the Lightweight CCM platform mapping re-
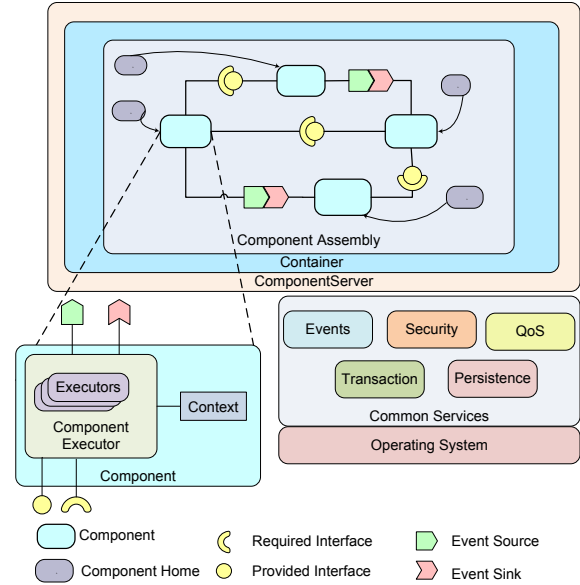

Figure 2: Key Elements in the CORBA Component Model

quires code generation for various infrastructure elements, including the following:

• **Component context**. A component context class is generated for each component interface to allow component reuse in multiple execution contexts. For example, generation of the component context `ed_A_Context` enables reuse of the event detector `ed_A` across different operational strings of our shipboard application.

• **Component base interfaces**. Each component interface derives from a number of base interfaces, *e.g.* `SessionComponent` and `EntityComponent` in Lightweight CCM, which classify the category of a particular component.

• **Component home**. A component home is generated corresponding to each component interface. Each component home provides (1) factory operations that allow customization of creating components and (2) finder operations that clients use to locate a component managed by a component home. In our shipboard example, distinct component

homes (`ed_A_Home, sm_A_Home`) are generated corresponding to each type of component (`ed_A, sm_A`).

• **Navigation operations**. Each component also contains a number of pre-defined navigation operations. The navigation operations of a component interface allow clients of a component to query and obtain references to component ports in a standardized fashion.

In addition to the interfaces and operations described above, each component implementation is typically split into multiple shared libraries. For example, component implementations are often split into three shared libraries: (1) *stub library* (*e.g.*, `ed_A_stub`), which contains the automatically generated client-side proxy code necessary for each component type to connect to other component types, (2) *servant*(*e.g.*, `ed_A_svnt`, which contains automatically generated code that registers a component with an Object Request Broker (ORB), and (3) *executor*(*e.g.*, `ed_A_exec`), which contains the business logic of a component written by application developers.

The drawbacks of designing DRE systems using multiple shared libraries are well-known [6] and include increased (1) code size, (2) dependencies between shared libraries, and (3) number of relocations at load time, which result in increased dynamic memory footprint. The overhead due to the static footprint increases with the increase in the number of component types. This overhead can be significant in complex applications and becomes apparent in the presence of a large number of types or in resource-constrained environments, which are common in DRE systems.

2.3. Dynamic Footprint Overhead

The code generated per component interface that allows containers to host components creates a number of auxiliary middleware infrastructural elements corresponding to each component instance at run-time, including:

• **Component home**. In Lightweight CCM, a component home can manage only one type of component. The run-time infrastructure, therefore, creates a separate component home instance for every component type loaded into a system, which is then used to create multiple component instances. Naïve implementations could also create a component home instance per component instance. Lightweight CCM also allows clients to create components dynamically by obtaining a reference to its component home. In many DRE systems, these sophisticated features of component homes are rarely used and impose additional time/space overhead corresponding to each component instance created at run-time. For example, creation of a separate component home for the components in Figure 1a is redundant and wasteful since they share the same address space and have no interactions with other components.

• **Component context**. The Lightweight CCM run-time infrastructure creates a component context corresponding to each component instance that is deployed. Such component contexts increase the dynamic footprint in proportion to the number of component instances.

• **Component servant**. Each component instance must also be registered with the underlying middleware infrastructure to communicate with other components. Corresponding to each component instance, a component servant is created at run-time to register it with the Lightweight CCM middleware. Although component servants are critical to the functioning of a component, each component servant thus created increases the dynamic footprint of the system.

As described above, each component instance consumes a certain amount of memory in the run-time environment. For DRE systems with many components, it is imperative to reduce the number of component instances/types to reduce the memory consumption of the whole system. Although static footprint overhead of a component increases the minimum memory required by the component, it does not increase with the increase in the number of component instances on a single node. Dynamic footprint overhead, in contrast, increases linearly with the increase in the number of component instances. In a large-scale DRE system with thousands of components, reducing the dynamic overhead is essential to reduce the memory footprint requirement of the integrated system. Section 3.1 describes how our model-driven optimization techniques help to automatically reduce the total number of components in the system.

3. Deployment-time Optimization Techniques

As described in Section 2, a key source of footprint overhead is the number of peripheral infrastructure elements, such as component home and component context, created for each monolithic component. An approach that reduces the number of components deployed should reduce the number of peripheral infrastructure elements, thereby reducing the static and dynamic footprint of the component-based DRE systems. The approach presented in this paper uses deployment-time optimization techniques.

This section first describes the model-driven optimization techniques that help reduce the space overhead in large-scale component-based DRE systems using our shipboard application as the motivating example. It then presents the structure and functionality of the *Physical Assembly Mapper* (PAM), which is our model-driven tool that automates deployment-time optimization techniques in the context of Lightweight CCM.

3.1. Deployment-time Optimization Algorithms

As shown in Figure 1b, the central theme of our component assembly optimizations is the notion of "fusion." Fusion involves merging multiple elements into a semantically equivalent element. Key differences between the optimization techniques described in this section include (1) the scope at which the fusion is performed, and (2) the

rules governing which elements are fused. The optimization technique described in Section 3.1.4 fuses multiple components into a single physical assembly at the level of a single deployment plan (usually corresponding to a single operational string). The technique described in Section 3.1.5 also fuses components into a single physical assembly, but the scope of such fusion spans an entire application.

3.1.1. Terminology. We now define the key terms used in our physical assembly fusion algorithms. A *node* is the physical machine on which one or more components are deployed. A *domain* is the target environment composed of independent nodes and their inter-connections. A *collocation group* is defined as the set of components that are deployed in a single process of a target node. Each collocation group corresponds to a single OS process and is always associated with one target node. A *deployment plan* is a mapping of a configured system into a target domain. It serves as the blueprint used by the middleware to deploy an application; an application could be composed of one or more deployment plans. A *facet* is a distinct named interface exported by a component. A *receptacle* is an interface required for the proper functioning of a component; receptacles enable a component to use the functionality provided by facets of other components. A *publisher* is an event source defined by a component, and a *consumer* is an event sink defined by a component which subscribes to events published by other components.

Our fusion algorithms perform a series of checks to evaluate "fusion," *i.e.*, whether multiple elements (such as components) can be merged into a single element. The set of checks performed to evaluate if two components can be fused is domain-dependent, *i.e.*, each component programming model—along with the application semantics—defines the exact set of criteria used to determine "fusion." As applied to CCM, two components can be fused if there are no clashes in the names of each component's *facet, receptacle, publisher and consumer* ports. Our fusion algorithms are explained in detail in [7].

3.1.2. Assumptions and Challenges in Component Fusion. A physical assembly is defined as the set of components created from the monolithic components that are deployed onto a single process of a physical node. As shown in Figure 1b, our optimization techniques creates one or more physical assemblies (*e.g.* e1_A_sm1_A) by *fusing* monolithic components (e1_A and sm1_A) deployed into the same process on each node of the target domain. Our approach assumes that (1) physical assembly creation should not require changes to the existing implementations of monolithic components and (2) physical assembly creation should not impact existing clients of fused components.

At the core of our component fusion technique is the capability to fuse multiple components into a single physical assembly. Components interact with other components using ports. Fusing multiple components into a single component requires merging the ports of all the individual components. There are, however, the following challenges in fusing multiple components into a single physical assembly in a DRE system:

1. Ports of a component are identified using their names. Each component interface defines a namespace; each port kind (*e.g.*, facets, receptacles) defines its own unique namespace within a component. Port names are also used to locate the services provided by each component and affect the middleware glue code generated for each component. Since ports are the externally visible points of interaction, port names of a component must be unique within each component's corresponding port kind namespace. Any merging of multiple components into a single component must maintain this invariant. Section 3.2.1 describes how we address this challenge in PAM.

2. Each component exposes its externally visible state through its component attributes. When fusing multiple components together, it is necessary to ensure that the states (*e.g.*, attributes) of individual components are maintained separately. It is also necessary to allow clients to modify this state. Section 3.2.1 describes how we address this challenge in PAM.

3. Each component must be identified uniquely. To obtain the services of a component through its ports, clients must be able to locate the component via directory services, such as the CORBA Naming Service, LDAP servers, and Active Directories. If multiple components are fused into a single component, the external clients should still be able to look up the individual components using their original names. Section 3.2.1 describes how we address this challenge in PAM.

4. Each component relies on being supplied a component context. A component context is needed to connect the component with the services of other components (*e.g.*, to connect a receptacle of a component with the facet of another component) that it depends upon. If multiple components are fused together, each component in the fused physical assembly must be provided with a context that is compatible with each monolithic component's context since we do not want to change the original monolithic component's implementation. Section 3.2.2 describes how we address this challenge in PAM.

3.1.3. Common Characteristics of Fusion Algorithms. The property of fusion of two elements is non-transitive, *i.e.* if component a can be fused with component b, and component b can be fused with component c, component a cannot always be fused with component c. For example, in Fig-

ure 1a, although component `p2_A` can be fused with `p1_A` and `p1_A` can be fused with `p3_A`, `p2_A` cannot be fused with `p3_A` since they both have a event sink port called `track`. Every pair of components must therefore be examined to determine if they can be fused together.

If *n* is the number of candidate components for each algorithm, *e.g.*, set of components deployed in a single process, *k* is the number of components that result from merging components together, then the number of components will be reduced by $\frac{n-k}{n}$. Of the components that can be fused into a single element, our goal is to find the largest set of elements since the larger the number of components that we can fuse together, the greater the reduction in the number of components. The best case is when $k = 1$, *i.e.*, when we can fuse all components into a single monolithic components; the savings in this case will be $\frac{n-1}{n}$.

Given an undirected graph $G = (V, E)$, where V is the set of candidate elements, and E is the set of edges such that if two elements are connected then they can be fused together, the problem of finding the largest set of elements that can be fused together is equivalent to the problem of finding a maximum clique in the undirected graph G. A *clique* is a complete subgraph of G. A *maximum* clique is the largest clique in the graph. Figure 3 shows the different component types (each shaded differently) from our example application. The set of components $\{co\_A, p1\_A, p3\_A, ec\_A\}$ in Figure 3 form a maximum clique. If a clique is not a proper subgraph
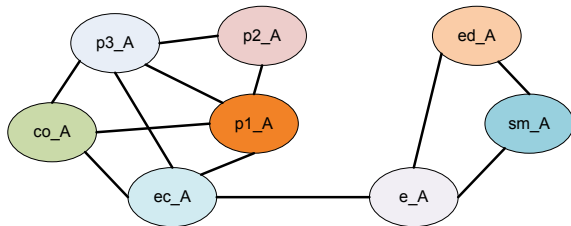

Figure 3: Maximal vs. Maximum Clique

of another clique, it is called a *maximal* clique (as opposed to maximum), *i.e.*, a maximal clique cannot be extended to create a larger clique by adding vertices to it. A maximum clique is also a maximal clique but the converse is not always true. The set of components $\{ed\_A, sm\_A, e\_A\}$ form a maximal clique in Figure 3. The maximum clique determination problem is NP-complete [8]. A maximum clique can be found by enumerating all the maximal cliques and choosing the largest. An efficient algorithm for enumerating the maximal cliques is by Bron and Kerbosch [9]. The worst-case time complexity for enumerating all maximal cliques has recently been proven [10] to be $O(3^{n/3})$, where *n* is the number of vertices in the graph.

We chose to trade-off the time savings by calculat-

ing just maximal cliques, *e.g.*, the set $\{ed\_A, sm\_A, e\_A\}$ in Figure 3 and using these to create physical assemblies, over the benefits of the footprint savings from creating physical assemblies out of maximum cliques, *e.g.*, the set $\{co\_A, p1\_A, p3\_A, ec\_A\}$. We calculate maximal cliques using a variation of the algorithm by Bron and Kerbosch [11], which has the desirable property that it enumerates the larger maximal cliques first. As shown in Section 4, our tests of this algorithm found that the maximal cliques it chose tend to also be maximum cliques. This result, however, need not hold true for all systems.

We developed two versions of the component fusion algorithm—*Local Component Fusion* and *Global Component Fusion*—that differ in the scope at which they are applied. As described below, the "local" version of the algorithm operates at the level of a single deployment plan (typically a single operational string), whereas the "global" version of the algorithm operates at the level of an entire application. Both algorithms assume that all high-level deployment planning (*e.g.*, resource allocation) is complete and the set of associations of components to nodes is finalized.

3.1.4. Local Component Fusion Algorithm. Small-scale DRE systems often use a single deployment plan to deploy an entire application, whereas large-scale DRE systems are usually deployed using multiple deployment plans. The local fusion algorithm initially collects the list of components that are deployed onto the different collocation groups (possibly on multiple nodes) and creates physical assemblies from the set of components that are local to that deployment plan.

Construction of physical assemblies by calculating maximal cliques directly on the graph of component instances will lead to significantly worse footprint compared to the original footprint. Since each clique results in a physical assembly, *i.e.*, a new component type, if component instances were used to calculate cliques, it will result in the creation of a number of new component types far exceeding that of the original set of types. For example, let us assume that we have three component types A, B and C with three instances each of A *(a1, a2, a3)*, B*(b1, b2, b3)*, and two instances of C*(c1,c2)*. Now suppose A and B can be fused together, and A and C can be fused together as well. If we apply the clique algorithm directly on the instances, we might end up with four types (AB, AC, B, C) corresponding to instances, say *a1b1, a2c1, a3b2, b3, c2*. Thus, we have increased the footprint by adding two new types (AB and AC) which have code common to A, B and C, but which cannot be shared since AB and AC are different types than A, B or C. We might also end up a different assignment, *e.g.*, three types (AC, AB, B) corresponding to instances *a1c1, a2c2, a3b1, b2, b3*. Both of these alternatives, however, are worse since they introduce more types than the

original three types (static footprint) though they reduce the number of instances (dynamic footprint). Irrespective of the heuristic chosen to start the clique formation (e.g., vertices with maximum degree), it is not deterministic, *i.e.*, we are not guaranteed to end up with an optimal assignment.

To prevent increase in the footprint due to a component type being present both as a stand-alone as well as part of a physical assembly, the local fusion algorithm uses a domain-specific heuristic: instead of calculating maximal cliques directly out of the all component instances belonging to a collocation group, it creates a set of component instances that occur the same number of times. By grouping components based on occurrence count, the heuristic results in either all instances of a single component type to be fused into one or more physical assemblies, or none at all. In the previous example, if we were to group components based on occurrence count, it would result in two component types: one new physical assembly (a new component type) AB with three instances *a1b1, a2b2 and a3b3* all sharing a single implementation and the type C with instances *c1, c2*. We, thus, have one fewer type (a single AB rather than both A and B) than the original and three fewer instances than the original. Moreover all the AB instances will end up sharing code and data segments in a Copy-On-Write (COW) fashion. Figure 4 shows the result of applying this heuristic followed by the "fusion" checks on the operational string from Figure 1a. Component instances of the same type are shaded using the same color in Figure 1a. Figure 1b shows the result of ap-
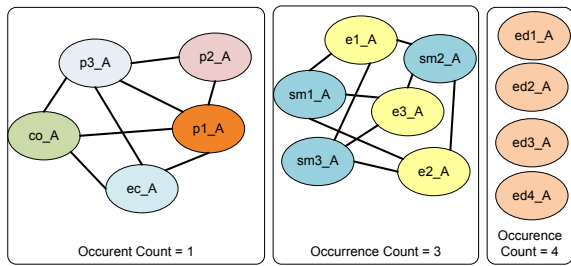

Figure 4: Input Graphs for Clique Determination

plying the local fusion algorithm on the components of Figure 1a. The algorithm creates five physical assemblies, `e1_A_sm1_A`, `e2_A_sm2_A`, `e3_A_sm3_A`, `ec_A_p2_A`, `co_A_p3_A`, merging all instances of `e_A` with `sm_A`, as well as merging `ec_A` with `p2_A` and `co_A` with `p3_A`. Further, none of the `ed_A` instances are fused with other component instances (*e.g.*, `p1_A` or the physical assembly `co_A_p3_A`) since not all `ed_A` instances can be fused.

3.1.5. Global Component Fusion Algorithm. The global component fusion algorithm is similar to the local algorithm, except that it operates across a set of deployment plans. The global algorithm can thus find more opportunities for creating physical assemblies. Section 4 measures and analyzes the benefits of the global component fusion algorithm versus the local algorithm.

3.2. Design and Functionality of the Physical Assembly Mapper

The algorithms described in Section 3.1 are sufficiently complicated that performing them manually is infeasible for large-scale DRE systems. Likewise, automating these algorithms by writing *ad hoc* scripts results in a brittle toolchain due to the complexity of the information, such as the interface definition files of the components and deployment metadata (*e.g.*, deployment plans and QoS configuration files), needed to perform these optimizations. We therefore used a more powerful technique—Model-Driven Engineering (MDE) [12]—to handle and optimize this information in a unified fashion.

To demonstrate our MDE optimization techniques, we developed a prototype optimizer called the *Physical Assembly Mapper* (PAM). PAM builds upon our previous work on the *Platform-Independent Component Modeling Language* (PICML) and *Component QoS Modeling Language* (CQML) [13] to implement the fusion algorithms described in Section 3.1. We implemented PAM as a *model interpreter*, which is a DSML-specific tool written in C++ for use with GME. Figure 5 presents an overview of the optimization process performed by PAM.
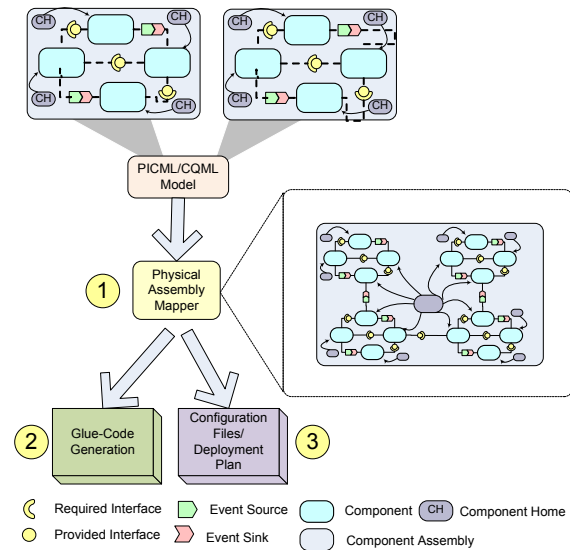

Figure 5: Workflow of the Physical Assembly Mapper

Component assembly optimization using PAM consists of three phases: (1) a model transformation phase described in Section 3.2.1, (2) a glue-code generation phase described in Section 3.2.2, and (3) a configuration files generation phase described in Section 3.2.3. Below we describe how each phase solves the challenges described in Section 3.1.2.

3.2.1. Model Transformation in PAM. The input to PAM is a model that captures the application structure and the QoS configuration options. This input model contains information about the individual component interface definitions, their corresponding monolithic implementations, collections of components connected together in a system-specific fashion to form virtual assemblies, associations of components with QoS configuration options. PAM implements both the local and global component fusion algorithms, which rewrite the input model into a functionally equivalent deployment model as shown in step 1 of Figure 5. As part of this model transformation, PAM creates physical assemblies, including interface definitions for the physical assemblies. Since the algorithms perform a series of checks before merging components together, PAM solves Challenge 1 of Section 3.1.2 (ensuring unique port names).

For each physical assembly PAM creates, it replaces the original set of component instances with an instance of the newly created physical assembly. This rewriting replaces all the connections to/from the original components with connections to/from the physical assembly. PAM also creates new attributes corresponding to each attribute of all the individual components, thereby ensuring no clashes with attribute names occur in the physical assembly namespace. PAM therefore solves Challenge 2 of Section 3.1.2 (maintaining the state of the individual components separately).

PAM creates configuration properties in the model associated with each physical assembly to enable clients to locate the original components via lookup services, such as the CORBA Naming Service, LDAP, and Active Directory. These configuration properties create multiple entries, one corresponding to each unique name used by the original components in lookup services, and ensures that all these names point to the physical assembly. PAM therefore solves Challenge 3 of Section 3.1.2 (uniquely identifying the components that comprise a physical assembly).

3.2.2. Generation of Glue-code in PAM. Once the model has been rewritten into a functionally equivalent optimized model, PAM utilizes a number of model interpreters to generate various artifacts related to the middleware glue-code as shown in step 2 of Figure 5. This middleware glue-code is necessary to use the physical assemblies created in the model with the existing monolithic implementations of the components. The glue-code generated by PAM creates a composite context by inheriting from the individual contexts of the components that make up the physical assembly. This derived context is compatible (due to inheritance) with each monolithic component's context and can be supplied to the individual component implementations at runtime by the container.

The glue-code generated for the physical assemblies can be compiled and deployed with the implementations of the other components in the system. PAM therefore solves Challenge 4 of Section 3.1.2 (providing a compatible context to the original component implementations). Since PAM performs the generation without requiring modifications to individual component implementations, PAM also achieves our goal of not imposing a burden on component developers by requiring changes to the original implementation.

3.2.3. Generation of Configuration Files in PAM. In addition to the middleware glue-code, PAM also generates modified metadata, such as deployment plans and QoS policy configuration files as shown in step 3 of Figure 5. When the local fusion algorithm is applied, PAM generates deployment plans in which the components that have been fused to form physical assemblies are replaced with the physical assemblies. All references to the original components are also replaced with references to the physical assemblies. The replacement of components (and their references) is done at the scope of a single deployment plan by the implementation of the local fusion algorithm in PAM.

When the global fusion algorithm is applied, PAM generates a single deployment plan. Since the optimizations are applied at the scope of the entire application, PAM merges the different deployment plans to create a single aggregate deployment plan. PAM then replaces the original components fused together to form physical assemblies, including the replacement of references as was done for the local fusion.

4. Empirical Evaluation and Analysis

To evaluate the benefits of our fusion algorithms described in Section 3.1, we applied PAM to the shipboard computing application described in Section 2.1. This section describes the characteristics of the application, explains our testbed setup, and presents the results of experiments that evaluate the footprint improvement from using PAM. Our experiments compare the space properties of applications developed using standard Lightweight CCM configurations against the execution of these applications after applying PAM to optimize the application.

4.1. Experimental Setup

We used 5 blades running Windows XP SP2 from the ISISlab (www.dre.vanderbilt.edu/ISISlab) open testbed for experimentation on DRE systems. Our experiments used version 0.5.10 of CIAO running on Windows XP SP2 and Linux with Ingo Molnar's real-time
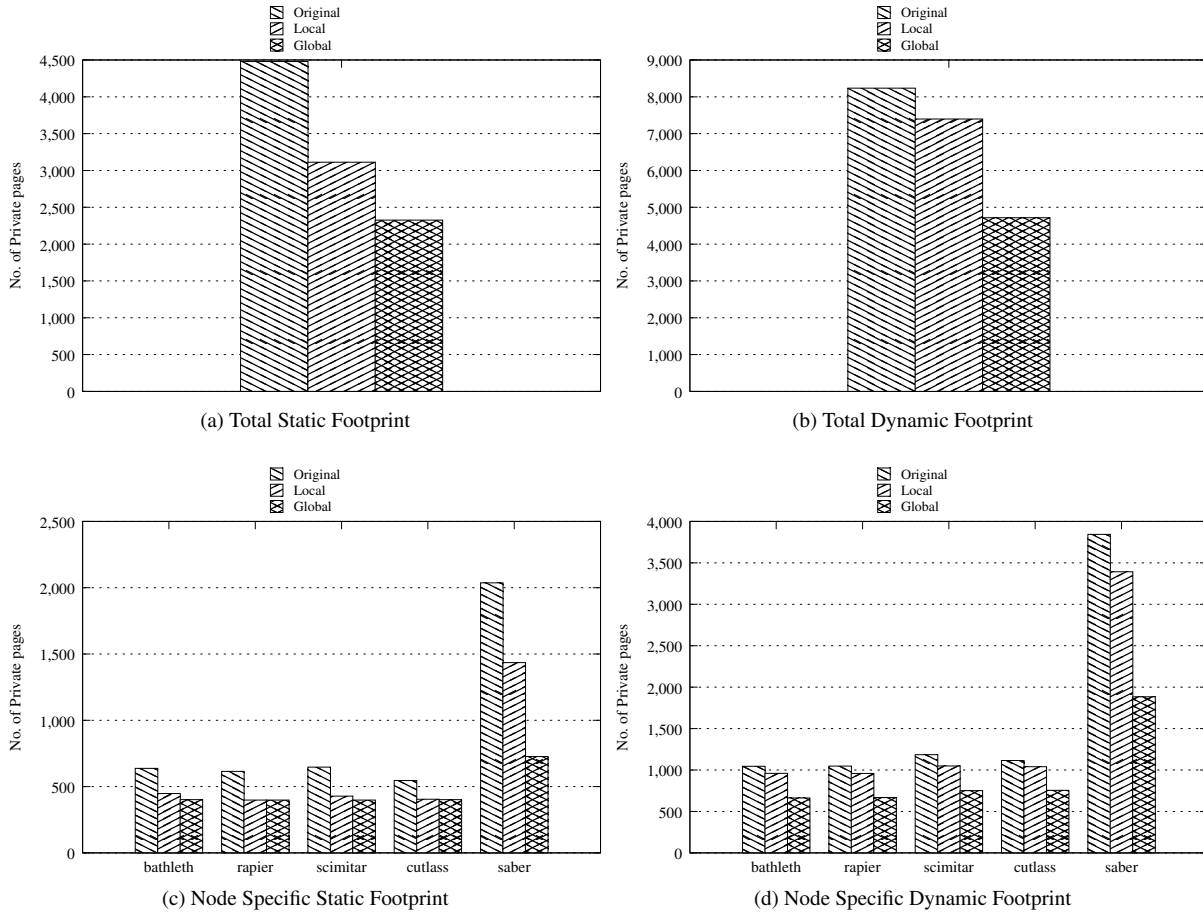
(a) Total Static Footprint

(b) Total Dynamic Footprint

(c) Node Specific Static Footprint

(d) Node Specific Dynamic Footprint

Figure 6: Static and Dynamic Footprint

pre-emption patches [14]. All machines were on the same local area network and were connected to each other using Gigabit ethernet. We measured the footprint of the components in the deployed shipboard computing application using Virtual Address Dump (VaDump) [15] to measure static (code and static data) and dynamic (heap memory) footprint of the components by taking a snapshot of the process that creates the container hosting the components on each machine.

4.2. Analysis of Empirical Footprint Results

Experiment design. To measure the footprint of the shipboard computing application, we deployed the 10 operational strings across the 5 nodes using 10 deployment plans. We allowed the application to execute for 5 minutes and measured the footprint of the components by running VaDump on the process hosting the components on each node. We refer to this run of the experiment as *Original* in all the graphs.

We used PAM (off-line) on the input model by invoking it to use the local component fusion algorithm described

in Section 3.1.4 and repeated the experiment using the 10 locally optimized deployment plans generated. We refer to this run of the experiment as *Local* in the graphs. We then used PAM (also off-line) on the input model by invoking it to use the global component fusion algorithm described in Section 3.1.5 and repeated the experiment using the single global deployment plan generated. We refer to this run of the experiment as *Global* in the graphs.

Analysis of results – Static Footprint. Figure 6a compares the static footprint, which includes the footprint contribution from code and the static data of the whole application deployed across all the 5 nodes. We measure the footprint of the application as the sum of the number of private and shareable (as opposed to shared) pages of the processes hosting the components using VaDump. The three runs of the experiment did not include the contributions from the OS and middleware shared libraries since they were unaffected by our optimizations.

As shown in Figure 6a, the original static footprint of the application was 4,478 pages and the application of the local

component fusion algorithm reduced it to 3,110 (a 31% improvement). Applying the global fusion algorithm reduced the static footprint further to 2,324 pages (a 49% improvement). The creation of physical assemblies by the component fusion algorithms therefore significantly reduced the static footprint of the application.

Analysis of results – Dynamic Footprint. Figure 6b compares the dynamic footprint of the application. The contributions here are primarily from the dynamic allocation of memory by the application in the three runs. VaDump does not provide the heap usage of individual shared libraries, so measuring the dynamic footprint of the application captures the heap usage of the whole process. Since we could not precisely pinpoint the heap usage of individual shared libraries, our dynamic footprint results are not as fine-grained as the static footprint results.

As shown in Figure 6b, the original dynamic footprint of the application was 8,231 pages. Applying the local fusion algorithm reduced it to 7,393 pages (an 11% improvement), whereas applying the global fusion algorithm reduced it to 4,713 pages (a 43% improvement). The reduction in dynamic memory stems primarily from reducing the number of homes and component contexts created in the physical assemblies. The increased reduction in the global compared to local is due to increased opportunities for creating physical assemblies (*i.e.*, the scope spans the entire application), as well as the merging of multiple deployment plans into a single deployment plan, which reduces the number of processes required to deploy the application.
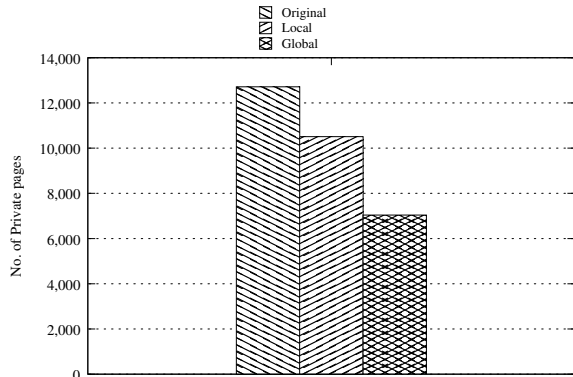

Figure 7: Total Footprint

Figure 6c and Figure 6d breakdown the total footprint across the different nodes. The increased footprint in the case of node saber in the three runs is due to the large number of components deployed on that node, *i.e.*, ~80 components compared to the average of 15-20 components on the other nodes. Figure 7 shows the combined (static+dynamic) footprint of the original application, which was 12,709

pages. The application of local fusion algorithm reduced the footprint to 10,503 pages (an 18% improvement), where as the application of the global fusion algorithm reduced it to 7037 pages (a 45% improvement).

5. Related Work

Optimizing middleware to increase the performance of applications has long been a goal of system researchers [16, 17]. This section compares our deployment-time optimizations to other component middleware optimization techniques. Optimization techniques to improve application performance can be categorized along the dimension of the time at which such optimization techniques are applied, *i.e.* design/development-time, run-time, or deployment-time. Our research in PAM is done at deployment-time.

Design/development-time approaches. Design-time approaches to component middleware optimization include eliminating the dynamic loading of component implementation shared libraries and establishing connections between components done at run-time, as described in static configuration of CIAO [18]. Our PAM approach is different since it uses models of applications to modify the structure of the assembly by creating physical assemblies, *i.e.*, new components, at deployment time. Our approach is therefore not restricted to optimizing just the inter-connections between components. Moreover, the static configuration approach can be applied in combination to our deployment-time optimizations.

Another approach to optimizing the middleware at design/development-time employs context-specific middleware specializations for product-line architectures [19], which exploits "invariant properties"—application-, middleware- and platform-level properties that remain fixed during system execution—to reduce the overhead caused by excessive generality in middleware frameworks. Researchers have also employed Aspect-Oriented Programming (AOP) techniques to automatically derive subsets of middleware based on use-case requirements [20], modify applications to bypass middleware layers using aspect-oriented extensions to CORBA Interface Definition Language (IDL) [21], synthesize middleware in a "just-in-time" fashion by integrating source code analysis, and inferring features and generating implementations [22].

The key difference between our approach in PAM and the various context-specific specializations and AOP-based techniques is that the optimizations performed by PAM do not require any input from the application developer, *i.e.*, the application developer need not design his application tuned for a specific deployment scenario. Our approach in PAM is, however, complementary to these approaches, since not all optimizations done via modification of appli-

cation advocated by these approaches are possible to perform at deployment-time using PAM.

Run-time approaches. Research on approaches to optimizing middleware at run-time has focused on choosing optimal component implementations from a set of available alternatives based on the current execution context. QuO [23] is a dynamic QoS framework that allows dynamic adaptation of desired behavior specified in *contracts*, selected using proxy objects called *delegates* with inputs from run-time monitoring of resources by *system condition* objects.

Other aspects of run-time optimization of middleware include 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 in PAM 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 efforts is to either build the middleware to satisfy certain performance guarantees, or effect adaptations via the middleware depending upon changing conditions at run-time. Our work in PAM is thus complementary to these approaches to application optimization.

Deployment-time approaches. Deployment-time optimizations research includes BluePencil [26], 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 work in PAM differs from BluePencil because it uses models of application structure and application deployment to serve as the basis for the optimization infrastructure. In contrast, BluePencil uses approaches like *configuration discovery* that extract deployment information from configuration files present in individual component packages. Since BluePencil operates at the level of individual client-server combinations, the kind of global optimizations performed by PAM are non-trivial to perform in BluePencil. BluePencil also relies on modification of the application source code to rewrite the application code, while PAM is non-intrusive and does not require application source code modifications.

6. Concluding Remarks

This paper described a model-driven approach to performing *deployment-time* optimizations. Our approach includes a family of optimization techniques that use *fusion* (*i.e.*, combining multiple elements into a single element) to reduce the number of elements without affecting the original semantics. We described two algorithms—Local Component Fusion, Global Component Fusion—that differ in the scope at which they operate.

We implemented the two algorithms in a prototype model-driven tool called Physical Assembly Map-

per (PAM), which is a DSML that supports development and optimization of component-based DRE systems using the Lightweight CCM. We conducted experiments on applying the techniques implemented in PAM on several representative DRE systems. Our results indicate that the PAM's deployment-time optimization techniques provide 45% improvement in footprint compared with conventional component middleware technologies. The following is a summary of lessons learned thus far from our work developing and applying PAM to optimize component-based DRE systems at deployment-time:

• **Deployment phase should be treated with equal importance.** The presence of a separate, well-defined deployment phase in DRE system development helps defer key system decisions to an intermediate stage between the traditional design/development-time vs. run-time. By using information available at deployment-time (but not available at design/development-time and that is too late for use at run-time), the deployment phase opens up new possibilities for system optimizations. In addition to system optimizations, deferring key system decisions until deployment-time helps increase reuse by decoupling deployment-time variability from component functionality.

• **Application-specific optimizations are critical to building large-scale systems.** While general-purpose optimizations can improve the performance of all systems, application- or context-specific optimizations have even more potential. By performing the optimizations in an application-specific fashion, we can obtain the benefits of these optimization techniques without the overhead of maintaining state or run-time evaluation in the middleware. Large-scale DRE systems thus start exhibiting an interesting inversion of the traditional process: instead of the application conforming to middleware characteristics, the middleware needs to conform to application characteristics.

• **Optimizations should be performed across layers in any layered architecture.** Our results show that any optimizations performed on a system with a layered architecture can significantly benefit from propagation of context information freely across the different layers. In addition to the propagation of deployment information to the middleware, we need to be able to propagate information from levels *above* the application deployment (*i.e.*, application functionality) and *below* the middleware (*i.e.*, operating system and system hardware).

PAM, PICML, and CQML are open-source and available for download at `www.dre.vanderbilt.edu/cosmic`.

References

[1] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *IEEE Real-time and Embedded Technology and Applications Symposium*, (Washington, DC), IEEE Computer Society, May 2003.

[2] P. Lardieri, J. Balasubramanian, D. C. Schmidt, G. Thaker, A. Gokhale, and T. Damiano, "A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems," *Journal of Systems and Software: Special Issue on Dynamic Resource Management in Distributed Real-time Systems*, vol. 80, pp. 984–996, July 2007.

[3] R. van Ommering, "Building product populations with software components," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, (New York, NY, USA), pp. 255–265, ACM Press, 2002.

[4] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, pp. 44–51, November 2001.

[5] D. de Niz and R. Rajkumar, "Partitioning Bin-Packing Algorithms for Distributed Real-time Systems," *International Journal of Embedded Systems*, 2005.

[6] U. Drepper, "How to write shared libraries." people.redhat.com/drepper/dsohowto.pdf, Nov 2002.

[7] K. Balasubramanian, *Model-Driven Engineering of Component-based Distributed Real-time and Embedded Systems*. PhD thesis, Vanderbilt University, 2007.

[8] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, eds.), pp. 85–103, New York, NY: Plenum Press, 1972.

[9] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[10] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theoretical Computer Science*, vol. 363, no. 1, pp. 28–42, 2006.

[11] I. Koch, "Enumerating all connected maximal common subgraphs in two graphs," *Theoretical Computer Science*, vol. 250, no. 1-2, pp. 1–30, 2001.

[12] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[13] A. Kavimandan, K. Balasubramanian, N. Shankaran, A. Gokhale, and D. C. Schmidt, "Quicker: A model-driven qos mapping tool for qos-enabled component middleware," in *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, (Washington, DC, USA), pp. 62–70, IEEE Computer Society, 2007.

[14] I. Molnar, "Linux with Real-time Pre-emption Patches." people.redhat.com/mingo/realtime-preempt/, Sep 2006.

[15] Microsoft, "Virtual address dump." support.microsoft.com/kb/927229, December 2006.

[16] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: a flexible, optimizing idl compiler," in *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, (New York, NY, USA), pp. 44–56, ACM Press, 1997.

[17] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas, "An efficient component model for the construction of adaptive middleware," in *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pp. 160–178, Springer-Verlag, 2001.

[18] V. Subramonian, L.-J. Shen, C. Gill, and N. Wang, "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), pp. 252–261, IEEE Computer Society, 2004.

[19] A. Krishna, A. Gokhale, D. C. Schmidt, J. Hatcliff, and V. Ranganath, "Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures," in *Proceedings of EuroSys 2006*, (Leuven, Belgium), Apr. 2006.

[20] F. Hunleth and R. K. Cytron, "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)*, pp. 38–45, ACM Press, 2002.

[21] Ömer Erdem Demir, P. Dévanbu, E. Wohlstadter, and S. Tai, "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), pp. 25–35, ACM Press, 2007.

[22] C. Zhang, D. Gao, and H.-A. Jacobsen, "Towards just-in-time middleware architectures," in *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 63–74, ACM Press, 2005.

[23] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[24] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic, "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), p. 301, 2002.

[25] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Syst.*, vol. 23, no. 1-2, pp. 85–126, 2002.

[26] S. Lee, K.-W. Lee, K. D. Ryu, J.-D. Choi, and D. Verma, "Ise01-4: Deployment time performance optimization of internet services," *Global Telecommunications Conference, 2006. GLOBECOM'06. IEEE*, pp. 1–6, Nov 2006.