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— **Component middleware technologies are increasingly used to assembly large-scale distributed real-time and embedded (DRE) systems composed from many individual components. While this approach allows DRE system developers to avoid traditional stove-piped monolithic architectures and better reuse portions of their systems, component middleware technologies can also impose space overhead in terms of static/dynamic memory footprint. Without effective optimization techniques, therefore, a large class of DRE systems with stringent footprint requirements may not be able to use component middleware effectively.**

This paper provides four contributions to the study of optimization techniques for component-based DRE systems. First, we describe the challenges in designing component-based DRE systems and identify the sources of overhead in typical component-based DRE systems. Second, we describe a class of optimization techniques that are applicable during 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 at multiple levels, local (deployment plan-specific) and global (application-wide). Fourth, we evaluate the benefits of these optimization techniques empirically and provide analysis of 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.

I. Introduction

A. *Overview and Challenges of Component Middleware for DRE Systems*

Component middleware technologies, such as the Lightweight CORBA Component Model (CCM), Boeing's PRiSM, OpenCOM, nesC's component model, and Timing Definition Language extension to Giotto, have raised the level of abstraction used to develop distributed, real-time and embedded (DRE) systems, such as avionics mission computing [1] and shipboard computing systems [2]. In addition to elevating the level of abstraction, 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*). A component assembly is thus a set of monolithic components inter-connected using the ports of the components.

Although component middleware provides many benefits, a number of challenges may restrict its use for a large class of DRE systems with stringent footprint requirements. For example, while functional decomposition of DRE systems into component assemblies and monolithic components helps promote reuse across entire product lines [3], it can also increase the number of components in the system. These many components, in turn, can significantly increase the memory footprint of component-based DRE systems.

Enterprise systems can offset the increase in footprint via hardware upgrades. In contrast, DRE system domains, such as avionics mission computing and shipboard computing, often cannot afford this luxury. This difference between enterprise systems and DRE systems stems from operational QoS demands and from the relatively long lifetime (about 15 to 20 years [4]) of DRE systems.

There are significant challenges in identifying and optimizing component implementations in the absence of a well-defined description of the systems that can be processed by tools automatically. Performing such optimizations on components is hard since the usage of any single component tends to span multiple compositional hierarchies, *i.e.*, a single component could be connected to different sets of components in different assemblies, in any complex system. Since components are often reused across an entire product line, an optimization that is applicable in one context may not be applicable in another context.

It is therefore infeasible to perform these optimizations in isolation. Instead, they should be performed based on the requirements of every unique deployment. Without the ability to augment the information available at the implementation level with deployment information, it is not possible to optimize component implementations by operating at the middleware level alone. Finally, performing these optimizations manually is in systems with many components that change due to system evolution.

B. *Solution Approach → Physical Assembly Optimizer*

To address the challenges of large-scale component-based DRE systems described above, we have developed model-driven *deployment-time* optimization techniques that help reduce the overhead in DRE systems. Our optimization techniques 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. By combining together multiple components to create *physical assemblies* of components using a technique known as *fusion*, our optimizations reduce the number of components required to deploy a system.

Assemblies of components as defined by standard component middleware, such as Lightweight CCM, are *virtual*, *i.e.*, the individual components that form the assembly can be spread across multiple machines of the target domain. Monolithic components of virtual assemblies are mapped onto the target nodes of the domain as part of the deployment process. The fusion of multiple components creates a *physical* 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 of a physical node, as shown in Figure 1. A physical assembly

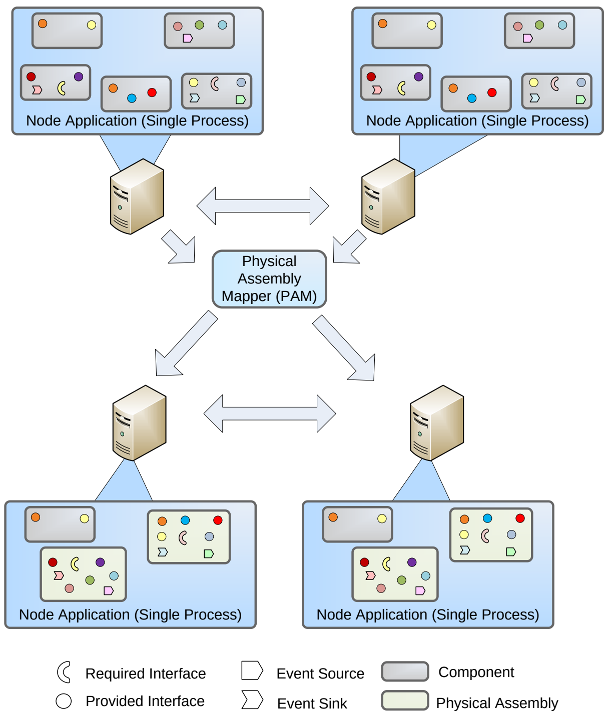

Node Application (Single Process)
Node Application (Single Process)
Physical Assembly Mapper (PAM)
Node Application (Single Process)
Node Application (Single Process)
Required Interface
Provided Interface
Event Source
Event Sink
Component
Physical Assembly

Fig. 1: Physical Assembly

is itself a full-fledged component, *i.e.*, it has a component interface as well as an implementation. The implementation of the physical assembly, however, simply delegates to the original implementations of the monolithic components from which the physical assembly is created.

A key enabler in the creation of physical assemblies is the presence of (1) *application structure information*, *i.e.*, connections between components, (2) *application QoS configuration information*, *i.e.* QoS configuration associated with each component, and (3) *application deployment information*, *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 even the domain of component middleware itself. The use of DSMLs to capture information across the different stages of DRE system development allows us to optimize the application at “deployment-time” in an application-specific fashion.

The novelty of our approach stems from identifying and applying component assembly optimizations in an opportunistic and automatic fashion from models of applications. This approach eliminates the difficulties associated with applying these optimizations manually. Such optimizations are infeasible to perform in a generalized manner at the middleware level due to the context-dependent nature of these optimizations.

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) [5], which is a meta-programmable environment for creating DSMLs. PAM utilizes both the connectivity information between components modeled and the QoS policies to create physical assemblies.

By operating at the high-level abstraction of models, PAM allows optimizing virtual 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 monolithic component implementations, functional decomposition, or structure of component-based DRE systems.

C. *Paper Organization*

The remainder of this paper is organized as follows: Section II motivates the need for PAM by surveying key sources of overhead in component middleware; Section III describes the deployment-time optimization techniques implemented by PAM; Section IV analyzes the results from empirical evaluation of DRE systems optimized using PAM on Lightweight CCM; Section V compares our work on PAM with related research; and Section VI presents concluding remarks and lessons learned.

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

This section describes key features of component middleware programming models and describes the cost of these features with respect to memory footprint for DRE systems. To make our discussion concrete, we use the Lightweight CORBA Component Model (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 [1], and OpenCOM [6].

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; 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.

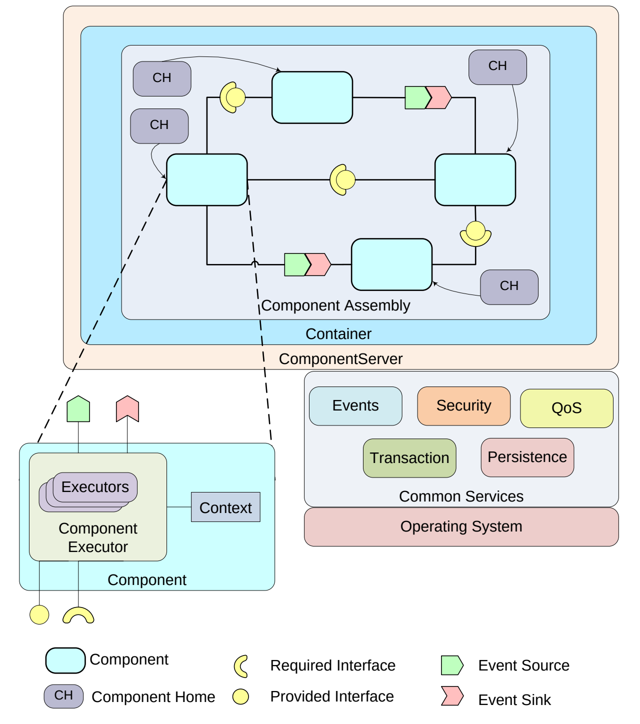


Fig. 2: Key Elements in the CORBA Component Model

A. *Static Footprint Overhead*

As shown in Figure 2, for every component type in a DRE system, the CCM platform mapping requires generation of code for various infrastructure elements, including the following:

• **Component context**. A component context class is generated corresponding to each component interface to allow each component to be reused in multiple execution contexts.

• **Component base interfaces**. Each component interface derives from a number of base interfaces, *e.g.* `SessionComponent` and `EntityComponent` in 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 not only factory operations that allow customization of creating components but also provides finder operations that clients use to locate a component managed by a component home.

• **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 the ports of a component in a standardized fashion.

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

The drawbacks of designing DRE systems using multiple shared libraries are well-known [7] and include increased code size, increase in the number of dependencies between shared libraries and number of relocations at load time, all of which result in increased dynamic memory footprint. Developers are thus forced to make a choice with respect to the granularity of the component functionality and implementations. The design trade-off is between (1) a *single monolithic shared library*, which can increase the footprint of components that only need to connect to it, compared to (2) a *number of shared libraries*, which can increase the overall footprint and time taken to load the libraries into memory. Section III-A describes how our optimization techniques defer this design decision until deployment-time, which is more flexible than making it a design/development-time. Deferring the decision until “deployment-time” allows for more opportunities to optimize the system using the extra information available only at deployment-time.

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.

B. *Dynamic Footprint Overhead*

The code generated per component interface that allows containers to host components adds to the static footprint and creates a number of auxiliary middleware infrastructural elements corresponding to each component instance at runtime, including:

• **Component home**. Since a component home can manage only one type of component, the CCM run-time infrastructure creates a separate component home instance for every component type loaded into a system. This component home is then used to create multiple component instances. Naïve implementations could also create a component home instance per component instance. CCM allows clients to create components dynamically by obtaining a reference to its component home. In many classes of DRE systems, these sophisticated features of component homes are seldom used and impose additional time/space overhead corresponding to each component instance created at run-time.

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

• **Component servant**. Each component instance must also be registered with the underlying middleware infrastructure to communicate with other components. A component servant is created at run-time corresponding to each instance of a component and allows it to be registered with the middleware. Although component servants are critical to the functioning of a component, each component servant created contributes to the increase in the dynamic footprint of the system.

Each component instance consumes a certain amount of memory in the run-time environment. In the presence of a large number of components, it is imperative to reduce the number of component instances/types to reduce the memory consumption of the system as a whole. To reduce the dynamic memory footprint of the system due to auxiliary middleware infrastructure elements, the designers are forced to make a decision with respect to the number of components as well as the granularity of the components during the creation of assemblies, *i.e.*, design/development-time. It is non-trivial to keep track of redundant component instances during component assembly creation, since each such component instance can be spread across multiple assemblies, *i.e.*, sub-systems.

Forcing the designer to pay attention to issues like number of component instances created and redundancy in component instances, during component assembly design distracts the designer from the high-level issues like functionality of the assembly. The design trade-off here is between (1) fine-grained decomposition of the system into a number of component types/instances, which can increase memory footprint, compared to (2) monolithic architectures that are strongly coupled, brittle, and discourage reuse, but which reduces the memory consumption of the auxiliary middleware infrastructure elements (by creating few of them). Section III-A describes how the creation of physical assemblies reduces the number of components in the system without requiring the use of monolithic architectures at design/development-time.

Although static overhead of a component increases its lower limit of the memory requirement, this type of overhead does not grow as the number of components increases on a single node. Dynamic overhead, in contrast, increases linearly as the number of components grows. In a large-scale scenario with thousands of components, reducing the dynamic overhead is essential to reduce memory footprint requirements of an integrated system. Section III-A describes how our optimization techniques reduce the total number of components in the system.

III. Deployment-time Optimization Techniques

As described in Section II, 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 time and overhead in large-scale component-based DRE systems. It then presents the structure and functionality of the *Physical Assembly Mapper* (PAM), which is a tool that automates deployment-time optimization techniques in the context of the CORBA Component Model (CCM).

A. Deployment-time Optimization Algorithms

As shown in Figure 3, the central theme of our component assembly optimizations is the notion of "fusion." Fusion

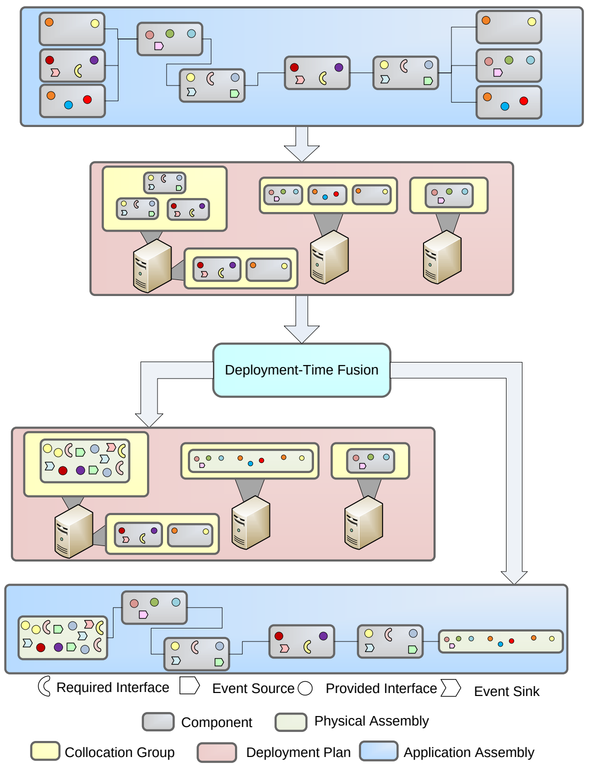


Fig. 3: Workflow of Deployment-time Fusion

involves merging multiple elements into a semantically equivalent element. Key differences between the various optimization techniques described in this section include (1) the type of elements fused, (2) the scope at which such fusion is performed, and (3) the rules governing which elements are fused.

The optimization technique described in Section III-A.4 fuses multiple components into a single physical assembly at the level of a single deployment plan; the technique described in Section III-A.5 also fuses components into a single physical assembly but the scope of such fusion spans an entire application.

1) 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. Our optimization techniques creates one or more physical assemblies by *fusing* monolithic components deployed into the same process on each node of the target domain. To ensure that our component fusion technique for creating physical assemblies does not degenerate to the static technique described in Section III, our approach operates under the following assumptions:

1) Physical assembly creation should not require changes to the existing implementations of monolithic components.
2) Physical assembly creation should not impact existing clients of fused components.

At the core of the component fusion technique is the capability to merge multiple components into a single physical assembly. Components interact with the external world 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 kind of port (*e.g.*, facets, receptacles etc.) 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 the corresponding port kind namespace. Although this holds true for each individual component, it need not be true when merging multiple components into a single component. Section III-B.2 describes how we address this challenge in PAM.
2. Each component relies on being supplied a component context. This context is needed to connect the component with the services of other components 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. Section III-B.3 describes how we address this challenge in PAM.
3. Each component maintains its externally visible state through its component attributes. When fusing multiple components together, it is necessary to ensure that the states of the individual components are maintained separately. It is also necessary to allow modification to such state from external clients. Section III-B.2 describes how we address this challenge in PAM.
4. Each component must be identified uniquely. To obtain the services of a component through its ports, external 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 III-B.2 describes how we address this challenge in PAM.

2) Common Characteristics of Fusion Algorithms: 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 property of fusion of two elements is non-transitive, *e.g.*, if component a can be merged with component b, and component b can be merged with component c, it doesn't always hold true that component a can be merged with component c. Every pair of elements must be examined to determine if they can be merged together.

If n is the number of candidate elements for each algorithm, *e.g.*, set of components deployed in a single process, k is the number of elements that result from merging components together, then the number of elements will be reduced by $\frac{n-k}{n}$. Of the elements that can be merged into a single element, our goal is to find the largest set of elements because the larger the number of elements that we can merge, the greater the reduction in the number of elements. The best case is when $k = 1$, *i.e.*, the savings 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 merged together, the problem of finding the largest set of elements that can be merged together is equivalent to the problem of finding a maximum clique in the undirected graph G. The maximum clique determination problem is well-known to be NP-complete [8].

A *maximal* clique (as opposed to maximum) is a clique that 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. One can find a maximum clique 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. Our component fusion algorithm, therefore, does not calculate maximum clique by enumerating all maximal cliques and choosing the largest.

For our first implementation, we chose to trade-off the time savings by calculating just maximal cliques (as opposed to a maximum clique) and using these to creating physical assemblies, over the benefits of the footprint savings from creating physical assemblies out of maximum cliques. We, therefore, use a variation of the algorithm by Bron and Kerbosch due to Koch [11] to calculate the maximal cliques. This algorithm has the desirable property that it enumerates the larger maximal cliques first. In our preliminary testing of the algorithm with some representative DRE systems, as shown in Section IV, we found that the maximal cliques chosen by our current algorithm tend to also be maximum size cliques. This, however, does not hold true for all systems. We intend to make the choice between maximal and maximum clique as an option to our tool that implements the algorithms.

3) Terminology: We now define some terms used in our 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 interconnections. 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.A deployment plan serves as the blueprint to be used by the middleware to deploy an application; an application could be composed of one or more deployment plans. The algorithms use several auxiliary functions described in detail in [12] and are briefly described below:

- $components(cg)$ Returns the set of components that belong to the collocation group cg.
- $types(I)$ Returns the set of types corresponding to the component instances in I.
- $collocationgroups(P)$ Returns the set of collocation groups that are defined in the deployment plan P.
- $nodes(P)$ Returns the set of nodes that are defined in the deployment plan P.
- $CreatePhysicalAssemblies(T, I)$ Creates physical assem-

blies from the set of components I whose types are described in T.
- $UpdateDeploymentPlan(IP, K)$ Updates the deployment plan IP by replacing all references to components with references to physical assemblies in K.

4) Local Component Fusion Algorithm: We developed two versions of the component fusion algorithm, both of which operate under the assumption that all high-level deployment planning (*e.g.*, resource allocation) has been completed and the set of associations of components to nodes is finalized. The two algorithms differ in the scope at which they are applied. Algorithm 1 is called *Local Component Fusion*, where "local" refers to the fact that this version of the algorithm operates at the level of a single deployment plan. Algorithm 2 is called *Global Component Fusion*, where "global" refers to the fact that this algorithm operates at the level of an entire application.

Algorithm 1: Local Component Fusion

```
Input: DeploymentPlan $IP$
Result: DeploymentPlan $OP$
begin
    CollocationGroup $cg$;
    Component $c$; set of Component $I$;
    ComponentType $t$; set of ComponentType $T$;
    set of set of Component $K$;
    foreach $cg \in collocationgroups(IP)$ do
        $I \leftarrow \{c \mid c \in components(cg)\}$
        $T \leftarrow \{t \mid t \in types(I)\}$
        $K \leftarrow K \cup$ CreatePhysicalAssemblies ($T$, $I$)
    end
    $OP \leftarrow$ UpdateDeploymentPlan ($IP$,$K$)
end
```

Smaller DRE systems might use a single deployment plan to deploy the whole 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.

Algorithm 1 uses a domain-specific heuristic to construct the set of components which are used to create physical assemblies.Instead of creating a clique directly out of the all component instances belonging to a collocation group, we create a set of component instances that occur the same number of times. Thus, the heuristic will result in selecting components that occur only once for the first invocation, , components that occur twice for the second invocation, and so on.

As a result of our heuristic, either all instances of a single component type are merged into one or more physical assemblies, or it is left alone. The algorithm never creates a component type that appears both in some physical assembly *and* stand-alone. Without this heuristic, the static footprint of the process will be significantly worse compared to the original footprint. The reason for this overhead is because we will load both the original implementation libraries of the component

Algorithm 2: Global Component Fusion

```
Input: set of DeploymentPlan $IP$
Result: DeploymentPlan $OP$
begin
    Node $n$; set of Node $N$; DeploymentPlan $p$;
    CollocationGroup $cg$;
    set of CollocationGroup $cgs$;
    set of set of Component $K$;
    Component $c$; set of Component $I$;
    ComponentType $t$; set of ComponentType $T$;
    foreach $p \in IP$ do
        $N \leftarrow \{n \mid n \in nodes(p)\}$
    end
    foreach $n \in N$ do
        $cgs \leftarrow \{cg \mid cg \in collocationgroups(n)\}$
        foreach $cg \in cgs$ do
            $I \leftarrow \{c \mid c \in components(cg)\}$
            $T \leftarrow \{t \mid t \in types(I)\}$
            $K \leftarrow K \cup$ CreatePhysicalAssemblies ($T$, $I$)
        end
    end
    foreach $p \in IP$ do
        $OP \leftarrow OP \cup$ UpdateDeploymentPlan ($p$, $K$)
    end
end
```

as well as the new physical assembly into the same process; components which end up being stand alone, as well as part of a physical assembly will contribute to the static footprint twice (or more if they are part of multiple physical assemblies).

5) Global Component Fusion Algorithm: The second version of the component fusion algorithm is called "Global Component Fusion" and is shown in Algorithm 2. "Global" refers to the fact that this version of the algorithm uses system-wide deployment information and is not constrained to a single deployment plan. The benefits of applying the algorithm at the global scope is measured and analyzed in Section IV.

The global fusion algorithm is similar to the local except that it operates across a set of deployment plans. The global algorithm can find more opportunities for creating physical assemblies. As shown in Figure 4, global fusion is different from local fusion since it merges all deployment plans of a DRE system, instead of updating the individual plans like the local algorithm.

B. Design and Functionality of the Physical Assembly Mapper

The algorithms described in Section III-A are sufficiently complicated that attempting to perform them manually will not scale for large-scale DRE systems. We can effectively rule out any manual attempt to perform these optimizations for large-scale DRE systems. The automation of the algorithms using existing methodologies like writing *ad hoc* scripts tools, results in a very brittle tool-chain. The main reason for the brittleness is that the vocabulary used to describe information such as the interface definition files of the components, the various deployment metadata like deployment plans, QoS configuration files necessary to perform these optimizations are disparate.

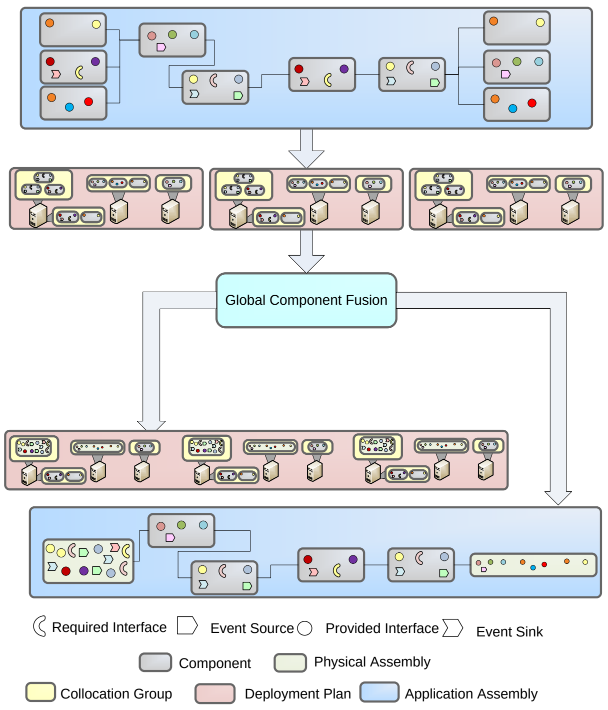


Fig. 4: Workflow of Global Fusion

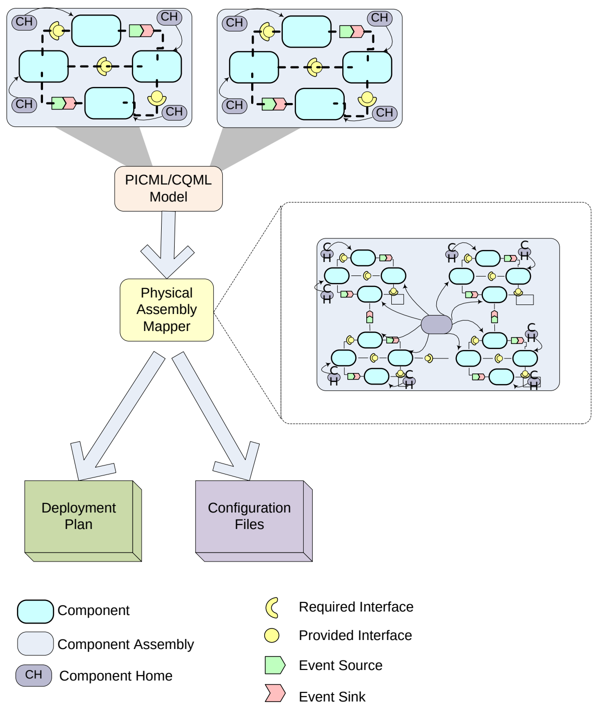


Fig. 5: Workflow of the Physical Assembly Mapper (PAM)

There is a need for a higher-level abstraction that allows dealing with these disparate sources of information in a unified fashion. Model-Driven Engineeringis a promising approach to providing this much sought after high-level abstraction.

1) Implementation of Fusion Algorithms in Physical Assembly Mapper: To demonstrate our optimization techniques, we developed a prototype optimizer called Physical Assembly Mapper(PAM), which builds upon our previous work on both *Platform-Independent Component Modeling Language* (PICML) and *Component QoS Modeling Language* (CQML) [13] to implement the fusion algorithms described in Section III-A. PAM is implemented as a model interpreter, a DSML-specific tool written using C++ for use with GME. Figure 5 presents an overview of the optimization process performed by PAM. Optimization using PAM consists of three phases: a model transformation phase described in Section III-B.2, a glue code generation phase described in Section III-B.3 and a configuration files generation phase described in Section III-B.4. Along with each phase, we also describe how the phase solves the challenges described in Section III-A.1.

2) Model Transformation in PAM: The input to PAM is the input model that captures the application structure and the QoS configuration options. The input model of the DRE system 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 Algorithm 1, the local component fusion and Algorithm 2, the global component fusion, to rewrite the input model into a functionally equivalent model. 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 deciding to merge components together, the issues with ensuring unique port names described in challenge 1of Section III-A.1 are non-existent when using PAM.

For each such physical assembly created, PAM replaces the original set of component instances with an instance of the newly created physical assembly. This writing 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 ensuring that there is no clash in the attribute names within the physical assembly namespace. Thus, PAM solves the problem with maintaining the state of the individual components separately described in challenge 3 of Section III-A.1.

To facilitate the lookup of the original components by external clients using mechanisms such as CORBA Naming Service, LDAP and Active Directory servers, PAM creates configuration properties in the model associated with each physical assembly. 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. Thus, PAM solves challenge 4 described in Section III-A.1.

3) 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. 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 run-time 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. Thus, PAM solves challenge 2 of Section III-A.1 associated with providing a compatible context to the original component implementations. Since PAM performs the generation without requiring modifications to individual component implementations, our original goal of not imposing a burden on the component developer by requiring changes to the original implementation is also achieved.

4) 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. When Algorithm 1 is applied, PAM generates deployment plans in which the components that have been merged 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 Algorithm 1 in PAM.

When Algorithm 2 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 merged together to form physical assemblies with the physical assemblies including the replacement of references as done for Algorithm 1.

IV. Empirical Evaluation and Analysis

To evaluate the benefits of our fusion algorithms described in Section III-A, we applied PAM on arepresentative DRE application, an application from the shipboard computing domain [2] This section describes the characteristics of the application, explains the experiment testbed architecture, presents the experiments to evaluate footprint improvement. Our experiments compare the space properties of applications developed using standard CCM configurations against the execution of these applications after applying PAM to optimize the application.

A. Experimental Platforms

1) Shipboard Application: The shipboard computing environment that we used for our experiments was developed using the CIAO middleware. This application consists of a number of components grouped together into multiple *operational strings*. As shown in Figure 6, an operational string is composed of a sequence of components connected together. Each operational string contains a number of sensors (*e.g.*, `ed1_A, ed2_A` shown on the far left) and system monitors (*e.g.*, `sm1_A, sm2_A` shown at the top) that publish data

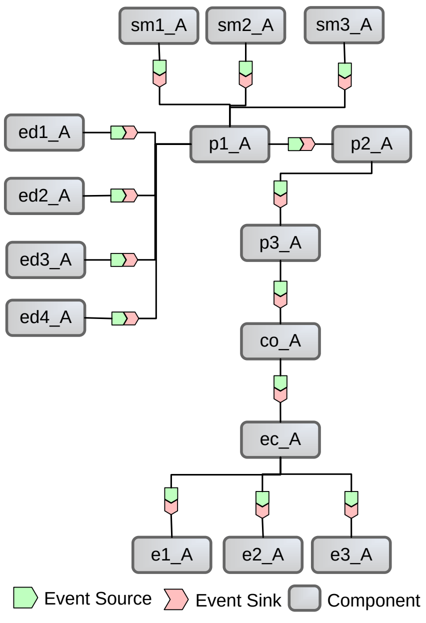


Fig. 6: Sample Operational String

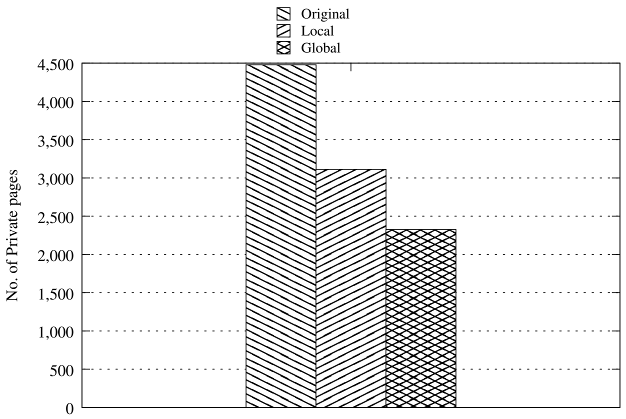
from the physical devices as well as the overall system state to a series of planners.

After analyzing the sensor data and the inputs from system monitors, the planners (*e.g.*,`p1_A, p2_A` shown in the center) perform control decisions using the effectors (*e.g.*, `e1_A, e2_A` shown at the bottom). Each operational string contains 15 components altogether, and the application used in our experiments is made up of 10 such operational strings, for a total of 150 components. Operational strings are at different importance levels. In case of a resource contention, the higher importance operational strings receive priority when accessing a resource.

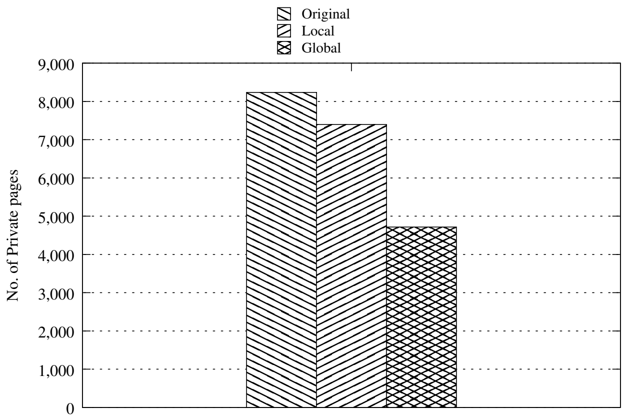
The application itself is deployed using 10 different deployment plans across 5 different physical nodes named bathleth, scimitar, rapier, cutlass, and saber. The assignment of components to nodes was determined *a priori* using high-level resource planning algorithms [14], and was available as input to our algorithms. Each node had a variable number of components, ranging from 20 to as high as 80 components assigned to it.

B. Experimental Setup

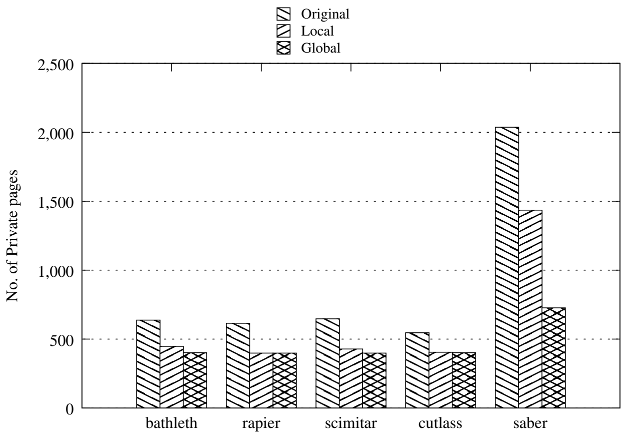
We used the ISISlab open testbed for experimentation on distributed real-time and embedded (DRE) systems and distributed continuous quality assurance (see `www.dre.vanderbilt.edu/ISISlab` for information on ISISlab). Our experiments used version 0.5.10 of CIAO running on Windows XP SP2 and Linux with Ingo Molnar's real-time preemption patches [15]. For the footprint experiments using the shipboard computing application described in Section IV-A.1, we used 5 blades running Windows XP SP2. All the machines were connected on the same local network and connected to each other using Gigabit ethernet. We measured the footprint of the components in the deployed shipboard



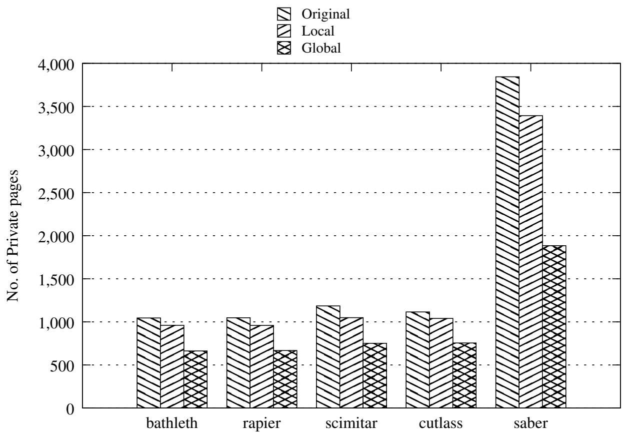
(a) Total Static Footprint



(b) Total Dynamic Footprint



(c) Node Specific Static Footprint



(d) Node Specific Dynamic Footprint

Fig. 7: Static and Dynamic Footprint

computing application using Virtual Address Dump (VaDump)distributed with the Windows Resource Kit Tools [16]. We used VaDump to measure both 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.

C. 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 the graphs shown below.

We used PAM (off-line) on the input model by invoking it to use the local component fusion algorithm described in Section III-A.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 shown below. Finally, we used PAM (also done off-line) on the input model by invoking it to use the global component fusion algorithm described in Section III-A.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 shown below.

Analysis of results – Static Footprint. Figure 7a 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 pages (as opposed to shared) of the processes hosting the components using VaDump. The three runs of the experiment did not include the contributions from the operating system and middleware shared libraries, since they were unaffected by our optimizations.

As shown in Figure 7a, 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, which is an improvement of 31%. Applying the global fusion algorithm reduced the static footprint further to 2,324 pages, which is an improvement of 49%. 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 7b 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. Unlike the static footprint measurements, measuring the dynamic footprint of the application captures the heap usage of the whole process, since VaDump does not provide the heap usage of individual shared libraries. 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 7b, the original dynamic footprint of the application was 8,231 pages. Application of the local fusion algorithm reduced it to 7,393 pages, which is an improvement of 11%. Application of the global fusion algorithm reduced it to 4,713 pages, which is an improvement of 43%. The reduction in dynamic memory stems primarily from reducing the number of homes and component context 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 is across 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 8 shows the combined footprint of the application. As shown in Figure 8, the combined footprint of the original application was 12,709 pages, the application of local fusion algorithm reduced it to 10,503 pages, which is an improvement of 18%. The application of the global fusion algorithm reduced it to 7037 pages, which is an improvement of 45%.

Figure 7c and Figure 7d provide the breakup of the total footprint across the different nodes. The increased footprint in the case of node saber in the three runs is due to the number of components deployed on that node.

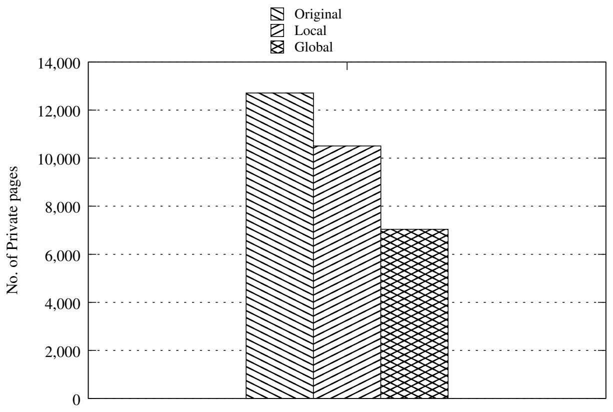


Fig. 8: Total Footprint

V. Related Work

Optimizing middleware to increase the performance of applications has long been a goal of system researchers [17], [18], [19]. In this section, we compare 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 [20]. 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 thus 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 [21], 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 [22], modify applications to bypass middleware layers using aspect-oriented extensions to CORBA Interface Definition Language (IDL) [23].

In addition, middleware has been synthesized in a "just-in-time" fashion by integrating source code analysis, and inferring features and synthesizing implementations [24]. 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 application 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 [25]. QuO [26] 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 domain-specific middleware scheduling optimizations for DRE systems [27], using feedback control theory to affect server resource allocation in internet servers [28] as well as to perform real-time scheduling in Real-time CORBA middleware [29]. 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 [30], 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. By operating 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.

VI. Concluding Remarks

Component middleware technologies, such as EJB and Lightweight CCM, allow developers to build DRE systems with a large number of components. The increase in the number of components imposes increased demands on DRE system resources. Without sophisticated tools and techniques for managing the complexity of large-scale component-based DRE systems, the benefits of using component middleware may be negated by the excessive resource demands. We therefore need optimization techniques in large-scale component-based DRE systems to ensure that the productivity benefits of component middleware are realized without compromising the overall requirements of the system.

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 Mapper (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 benefits provided by component middleware significantly alter the lifecycle of DRE system development with system deployment achieving importance similar to design, development and analysis/verification. 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 help 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. Our experiments with PAM show the footprint benefits of performing deployment-time optimizations in an application-specific fashion. An alternative approach is to perform these optimizations at run-time. For example, the middleware could try to fuse components to create physical assemblies dynamically at run-time instead of our application-specific deployment-time approach. Dynamic fusion of components would necessitate the middleware to keep state about the characteristics of the different components as well as evaluate the maximal clique algorithm at run-time. Such an approach, however, becomes infeasible in large-scale DRE systems due to the excessive state the middleware must maintain in addition to the evaluation of the fusion algorithms at run-time.

By performing the optimizations in an application-specific fashion, we can obtain the benefits of such fusion without the overhead of maintaining state or run-time evaluation. 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 indicate that irrespective of the number and kind of optimizations performed at the middleware layer, the middleware is ultimately restricted to the context information available to it. By using higher-level abstractions, such as DSMLs, we can perform optimizations that are not possible at the middleware layer alone.

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 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). Our approach currently only unifies two of these layers, so it must be extended to encompass all DRE system layers, *i.e.* the application, middleware, operating system and the underlying hardware.

Model-driven engineering has the potential to serve as the unifying foundation for building DRE systems. To separate the different phases of DRE system lifecycle—while also achieving the benefits gained from propagating information from one phase to another—it is necessary to create a pipeline for DRE system development. Just like a pipeline in a microprocessor relies on a common instruction set and allows processing instructions by splitting each instruction into a number of different stages, models and DSMLs can provide the basis for building a DRE system development pipeline. By using models to convey information across the different stages of such a DRE system development pipeline, we can realize the goals of exposing information across the different stages, thereby yielding DRE systems that are both optimized *and* easier to evolve.

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 *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, 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] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Software Product Lines: Experience and Research Directions*, vol. 576, (New York, NY, USA), Springer-Verlag, Aug 2000.

[5] 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.

[6] G. Coulson, G. Blair, M. Clarke, and N. Parlavantzas, "The design of a configurable and reconfigurable middleware platform," *Distributed Computing*, vol. 15, no. 2, pp. 109–126, 2002.

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

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

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

[15] I. Molnar, "Linux with real-time pre-emption patches." http://people.redhat.com/mingo/realtime-preempt/, Sep 2006.

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

[17] 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.

[18] 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.

[19] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet, "Specialization tools and techniques for systematic optimization of system software," *ACM Trans. Comput. Syst.*, vol. 19, no. 2, pp. 217–251, 2001.

[20] 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.

[21] 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), ACM, Apr. 2006.

[22] 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.

[23] 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.

[24] 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.

[25] A. Diaconescu, A. Mos, and J. Murphy, "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), pp. 214–221, IEEE Computer Society, 2004.

[26] 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.

[27] C. D. Gill, R. Cytron, and D. C. Schmidt, "Middleware Scheduling Optimization Techniques for Distributed Real-time and Embedded Systems," in *Proceedings of the 7th Workshop on Object-oriented Real-time Dependable Systems*, (San Diego, CA), IEEE, Jan. 2002.

[28] 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'02)*, (Washington, DC, USA), p. 301, IEEE Computer Society, 2002.

[29] 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.

[30] 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.