Evaluating Component Middleware Enhancements for Distributed Real-Time Embedded Systems

Nanbor Wang
nanbor@txcorp.com
Tech-X Corp

Venkita Subramonian and Chris Gill
{venkita,cdgill}@cse.wustl.edu
Dept. of Computer Science
Washington University

Doug Schmidt
schmidt@dre.vanderbilt.edu
Dept. of Computer Science
Vanderbilt University

Abstract

Component middleware enhances earlier generations of object middleware by dividing system development and configuration concerns into separate aspects such as application functionality, timeliness constraints, and resource access policies. However, conventional component middleware technologies, such as J2EE and .NET, were not designed to manage the stringent QoS aspects of distributed real-time and embedded (DRE) systems.

This paper makes four contributions to research on enhancing component middleware for DRE systems. First, it describes how the integration of Real-time CORBA object middleware features with CORBA Component Model (CCM) component middleware features in the context of the Component-Integrated ACE ORB (CIAO) improves the configurability of key real-time aspects. Second, it compares QoS performance and configuration flexibility of Real-time CORBA features with and without CCM features to measure the overhead of component middleware vs. object middleware. Third, it evaluates the performance of static vs. dynamic component configuration to support different types of DRE applications. Fourth, it presents an empirical comparison of component configuration in CIAO and PRISM, which is a domain-specific component model implementation developed for avionics systems by Boeing. Our results show that standards-based component model implementations can offer comparable performance to customized domain-specific component model implementations, while offering greater flexibility in configuring key DRE system aspects.

1 Introduction

Component middleware enhances earlier generations of object middleware by dividing system development and configuration concerns into separate aspects such as application functionality, timeliness constraints, and resource access policies. Furthermore, component middleware then allows those aspects to be specified, composed, and enforced at various system development lifecycle stages such as component packaging, application assembly, and system deployment.

Conventional component middleware technologies, such as J2EE and .NET, were designed to manage the quality of service (QoS) aspects of enterprise systems, which focus largely on scalability and transactional dependability. These conventional component middleware technologies are not well suited, however, to address the more stringent QoS aspects of distributed real-time and embedded (DRE) systems, which focus largely on low latency/jitter, timeliness, and fine-grain arbitration of access to shared resources. In particular, although conventional component middleware standardizes some mechanisms to configure and control certain QoS aspects such as connecting event sources to event sinks, or managing transactional behavior, it lacks effective abstractions for separating QoS policy configurations from application functionality, which forces application developers to configure QoS policies in an ad hoc way that is unnecessarily hard to configure, validate, modify, and evolve for complex DRE systems.

This paper makes four contributions to research on enhancing component middleware for DRE systems. First, it describes how the integration of Real-time CORBA object middleware features with CORBA Component Model (CCM) component middleware features in the context of the Component-Integrated ACE ORB (CIAO) improves the configurability of key real-time aspects, such as the priorities and periods of component method invocations. Second, it compares QoS performance and configuration flexibility of Real-time CORBA features with and without CCM features to measure the overhead of component middleware vs. object middleware. Third, it qualitatively and quantitatively evaluates the role of static and dynamic component configuration with respect to supporting different types of DRE applications. Fourth, it gives an empirical comparison of the configuration features in CIAO to the configuration features in PRISM, which is a domain-specific component model implementation developed for avionics systems by Boeing. Our results show that standards-based component model implementations can offer comparable performance to customized domain-specific component model implementations, while offering greater flexibility in configuring key DRE system aspects.

This paper is structured as follows. Section 2 describes how the integration of CIAO's configuration capabilities with TAO's Real-Time CORBA 1.0 policies and mechanisms allows fine-grain customization of real-time DRE system aspects at different stages of the application development lifecycle. Section 3 describes both the dynamic and static variants of CIAO's configuration infrastructure, and compares those variants to the static configuration infrastructure in the PRISM domain-specific component model. Section 4 describes and presents the results of a number of empirical studies we have conducted to quantify the performance benefits and costs of using CIAO to develop DRE systems. Section 5 describes related work in the areas of DRE system configuration tools and QoS-aware component models. Finally, Section 6 offers concluding remarks and describes future work.

2 Configuring DRE System Aspects with CIAO

To provision end-to-end QoS robustly throughout a component middleware system and to improve component reusability, QoS provisioning specifications should be decoupled from component implementations and specified instead in component composition meta-data. To achieve this decoupling, two major design challenges must be addressed at once: (1) customizing policies and mechanisms at fine granularity according to the specific requirements of each application, while (2) preserving sufficient flexibility to exploit different configuration options at different points in the development lifecycle, across a wide range of applications.

In this section we describe how CIAO can be used to custom-configure both the functional and QoS aspects of each application, within a reasonably wide range of applications with different real-time configuration needs. We focus on two key areas of interest for DRE systems built using standards-based COTS middleware, *i.e.*, CORBA: (1) which Real-Time CORBA 1.0 Object Request Broker (ORB) policies and mechanisms represent useful points of configuration to meet real-time application requirements, and (2) when in the overall system development lifecycle can these configuration points be exploited to improve customization of the system overall.

2.1 Real-Time CORBA Configuration Policies and Mechanisms

Within a real-time middleware endsystem, resources for enforcing real-time behaviors are allocated using various policy objects standardized by real-time middleware specifications such as Real-time CORBA. These policies can be applied with different granularities, *i.e.*, affecting different scope of objects. Depending on the purpose for which a policy is designed, it can be applied (1) to the client-side, *i.e.*, to affect how the ORB invokes remote operations, or (2) to the server-side, *i.e.*, to affect how the ORB handles incoming operation invocations, or (3) to both the client and server sides, *i.e.*, to control common mechanisms and strategies.

In Real-time CORBA, a policy can be applied at different scopes and granularities, according to the `Policy` management framework introduced by the CORBA Messaging specification [1, 2]. On the client-side, a policy can be applied with the following granularities to affect how the client ORB invokes remote operations:

1. **ORB.** Policies applied to an ORB apply to all object references resolved by the ORB thereafter. These policies affect all operation invocations using these object references.
2. **Thread.** ORB-level policies can be overridden, within a thread of execution, by applying the policy to the `PolicyCurrent` pseudo-object associated with the thread. All subsequent operations invoked from that thread of execution will be affected by the applied policies.
3. **Object Reference.** An object can apply different policies from the ORB or the context of the thread of execution by overriding the policies in the scope of an object reference. Subsequent operations invoked on that object reference will be affected by the overridden policies.

Likewise, a policy can be applied with various granularities on the server-side ORB to affect how the ORB handle incoming operations:

1. **ORB.** Policies applied to an ORB affects all servants hosted by the ORB, *i.e.*, all incoming requests handled by the ORB will be affected by the applied policies.
2. **POA.** ORB-level policies on the server side can be overridden by applying the policy to the POA. All incoming requests to the servants managed by the POA which the overriding policy applied will be affected by the policy.
3. **Object reference.** Certain POA-level policies, such as priority level, can be overridden in an object reference by specifying the overriding policies when activating or creating a new object reference or servant. This overriding mechanism allows a servant to handle incoming requests using different sets of policies.

It is important to note that in the CORBA policy management framework, policies applied to a method invocation override policies applied to the target object, which in turn override policies applied to the current thread of execution, which then override ORB-wide policies, which finally override default policies for that ORB implementation. In general, policies applied at finer levels of granularity override those applied with coarser granularity. This allows developers to configure an application with a set of default policies while being able to specialized parts of the application by overriding those default policies. It is therefore important that component middleware provide similar granularity in its configuration interfaces and mechanisms.

2.2 System Composition Phases

The CCM programming paradigm provides the foundation for composing systemic behaviors via specification of policies and mechanisms for a composed application. CIAO takes advantage of the multiple policy specification stages in the CCM development lifecycle to add hooks where systemic policies and mechanisms can also be specified, thus offering a significant advantage over the conventional Real-time CORBA development process. To achieve this, several supporting constructs in CCM have been extended in to support composition of systemic policies and mechanisms. To support real-time applications in CIAO, it is first important to identify when and how real-time policies and supporting mechanisms can be composed into an application.

Information about the structure and requirements of the application and about the resources on which is is to be deployed becomes available at different stages of the application's lifecycle. The following list describes how different kinds of QoS provisioning policies can be configured at various stages of the CCM development lifecycle and the consequences of using these composition strategies. Historically, policies for managing systemic behaviors have been integrated implicitly by developers during the development of the application programs themselves. Identifying the kind of systemic policies that should be composed into each stage of the development lifecycle provides better organization to configure and manage these policies.

1. **Component implementation stage.** In the component implementation stage, component developers can specify the policies and mechanisms on which a component implementation depends to execute correctly, *i.e.,* to meet the QoS requirements of the component implementation. For example, a developer may decide to manage the priority level a component uses to invoke operations on a particular receptacle in the component implementation explicitly. In this scenario, there needs to be a way for the component implementation to specify its dependency on component server and container that support real-time behavior.
2. **Component packaging stage.** A component implementation package may also document its key systemic behaviors as constraints in this stage. For example, a component may document its implementation constraints on allowable rates it can achieve to process or to propagate an incoming operation invocation from a facet to operations to receptacles.

   Binding systemic behaviors, such as Real-time CORBA's priority model policy, into component implementations makes these behaviors part of component implementations. This approach allows application assemblers to use different component implementations for selecting different systemic behaviors. However, as we described earlier, this approach may hamper the reusability of component implementations as they assume certain kinds of support from the runtime environment and other components that coexist with them. Moreover, extra care must be taken when composing components with embedded systemic behaviors to ensure all the components used to assemble an application have compatible systemic behaviors [3, 4].

   It is therefore important to extend component descriptors to allow developers to embed these implementation-specific dependencies and systemic behaviors. This extension will also provide hints to other tools to ensure that necessary supporting mechanisms are available in the composed application and that all the components have compatible behaviors.
3. **Application assembly stage.** During this stage, developers utilize various CASE design tools to create assembly specifications called *assembly descriptors* that describe how to build distributed applications using available component implementations. Information contained in assembly specifications includes the number of servers, what component implementations to use, how and where to instantiate components, and how to connect component instances together in an application. Policies and mechanisms for allocating resources and controlling systemic behaviors can be applied at this stage to control and to allocate resources used by assembled applications. These policies and mechanisms can be applied and associated with different entities via assembly descriptors, to provide fine-grained control over systemic behaviors.

   In addition, when component behaviors constraints are documented in component implementation packages, the application assembly stage allows application assembly tools to assimilate and reason with these constraints, make sure there are no conflicting constraints among component implementations, and deduce and synthesize a new set of constraints of the overall that are application. For example, if a series of component implementation are connected as a caller-callee chain via their facets and receptacles, with the embedded allowable rate constraints, the assembly tools will be able to deduce a new set of allowable rates for the new assembled application and embed the constraints in the assembly descriptors [4].
4. **Application deployment stage.** At the final stage of

transforming a component assembly into a fully specified and running application, component deployment tools are responsible to ensure the runtime environment, *e.g.*, the set of component servers, provides adequate support for the systemic behaviors the application demands. Support for systemic behaviors can either be provided by the deployment tools via a special component server implementation that offers the required mechanisms, or via dynamically linking in the required mechanisms into component servers. By controlling the systemic aspect support mechanisms, the deployment stage provides the last chance in the CCM development lifecycle to control how resources are allocated in the running application. Similar to component implementation, packaging and application assembly stages, tools can be used to model and assist the generation of deployment configuration to ensure the systemic requirements of applications can be met.

2.3 Composing Real-time Aspects with CORBA, CCM and CIAO

To address the dual challenges of customization and flexibility, component metadata representation and manipulation capabilities found in conventional component middleware must therefore be extended. In particular, they must be able to configure real-time policies at each of the different scopes listed in Section 2.1 and at each of the lifecycle stages listed in Section 2.2.

Systemic aspects tend to cross-cut functional boundaries. Composing systemic aspects often requires resources and mechanisms to be allocated and configured globally throughout an application. Therefore, the XML document formats for component and assembly metadata must be expanded to parse, allocate and configure these resources and associate them with component instances or component connections. Moreover, to ensure a component server is equipped with the mechanisms needed to support the provisioned QoS requirements, component packaging and assembly metadata should include middleware modules that enable the control and configuration of these resources.

We now summarize the strategies that can be applied to configure real-time properties of DRE applications using the Real-time CORBA features available in TAO and the QoS aspect configuration capabilities of CIAO. First, requiring a real-time ORB is not a policy of CORBA itself, but rather can be inferred from the presence of other real-time policies at any of the application lifecycle stages. If the priority model a component implementation will use will always be the same, it can be programmed directly into the component implementation; otherwise if a more flexible approach is appropriate, that decision can be made at the component packaging, application assembly, or even system deployment stages, and applied to the entire component or only particular interfaces, receptacles, or operations. Custom mappings of priorities specified by clients into priorities actually used on a server can be installed either at the component packaging or at the application assembly stage. Connections can be marked as private, i.e., not for re-use between other objects, either when facets and receptacles are specified or when those connections are established during application assembly. Thread pool sizes and priority-banded connection policies can be specified during component packaging or application assembly. Finally, connections can be marked for pre-establishment at system initialization during component packaging, application assembly, or system deployment.

Compared to using a traditional CORBA implementation to develop a simple client-server application, more steps are seemingly required to develop the same application using CCM, and CIAO adds even more steps for configuration of real-time aspects. However, much of the complexity seen in CIAO is inherent to the process of developing and deploying DRE applications, and in fact other *accidental complexities* that are addressed in CIAO can arise when developing DRE applications either using Real-time CORBA directly or using a component model implementation that does not provide explicit support for configuring real-time system aspects.

2.4 Example Application

We now examine a simple but illustrative example application drawn from the avionics mission computing domain [5]. Figure 1 illustrates a prototypical DRE application scenario involving three software entities: a **Rate Generator**, which wraps a hardware timer that triggers pushing of events at specific periodic rates to event consumers that register for those events, a **GPS** component which wraps one or more hardware devices for navigation, and a **Head-up Display**, which wraps the hardware for a display device in the cockpit. The GPS component caches its location value, which it refreshes from the navigation hardware device when it receives a triggering event from the Rate Generator Component. The GPS component then pushes a triggering event to the Heads-up Display component which in turn pulls the new value from the GPS component and updates its displays in the cockpit.

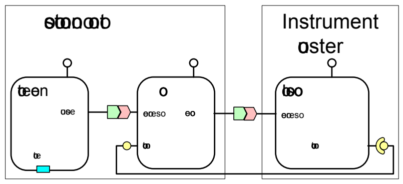

Figure 1: **A Prototypical DRE Application Scenario**

In practice, DRE applications are likely to involve a larger number of components, with subsets of components connected via specialized networking devices, such as VME-bus backplanes. However, although applications and their real-time requirements and deployment environments may differ, many DRE systems share the kinds of rate-activated computation and display/output QoS constraints illustrated here. Therefore, this example represents a broader class of systems to which our work on CIAO applies.

This example also helps to illustrate the benefits of developing DRE applications with CIAO, instead of with Real-time CORBA directly or with component model implementations that do not support explicit fine-grain configuration of key QoS aspects. With CIAO's extensions to the CCM development paradigm, extending the example application shown in Figure 1 is as easy as providing the new component implementations, packaging them with XML descriptors of their interfaces and QoS constraints, and then using those packages to compose new application via an XML assembly file. In contrast, extending a direct Real-time CORBA implementation of the same application would requires more effort in modifying code in each of the components, *i.e.*, in addition to creating the new servant implementations, to configure ORBs and POAs to accommodate the servants and their QoS requirements, activating the servants, setting up QoS attributes of the connections between the servants, and modifying how servants interact.

When it is necessary to extend an existing application due to changes in requirements or deployment platforms, the benefits of using CIAO's development model are magnified. For example, suppose a collision warning component and its associated cockpit display and rate generation components were added to the example application shown in Figure 1. Further suppose the collision warning system may run at a faster or slower rate than the navigation system depending on the kind of aircraft and its intended operating environment, but should always run at higher priority so that the pilot is always alerted of an impending collision in time to take an evasive maneuver (instead of worrying about the exact location of the aircraft).

ORBs conforming to Real-time CORBA specification allow developers to specify thread pool priorities and servant implementations can specify rates of invocation on other servants using mechanisms such as timers provided by the operating system. However, adding the code to set priorities and timer periods again involves intrusive modification of the servants themselves. In comparison, CIAO's real-time extensions to CCM provide new XML formats to define all the real-time parameters and policies in real-time extension files. These files can then be composed into an existing application assembly without modifying the existing component implementations.

3 CIAO Configuration Infrastructure

3.1 Dynamic Assembly of Components

The process of dynamic assembly of application components in CIAO is shown in Figure 2. The first stage of the CIAO system lifecycle occurs off-line, when component package (.csd) and assembly (.cad) files are generated by a modeling tool or other prior stage of the tool chain. These files contain an abstract specification of the configuration that is to be achieved by CIAO in each particular deployment environment.

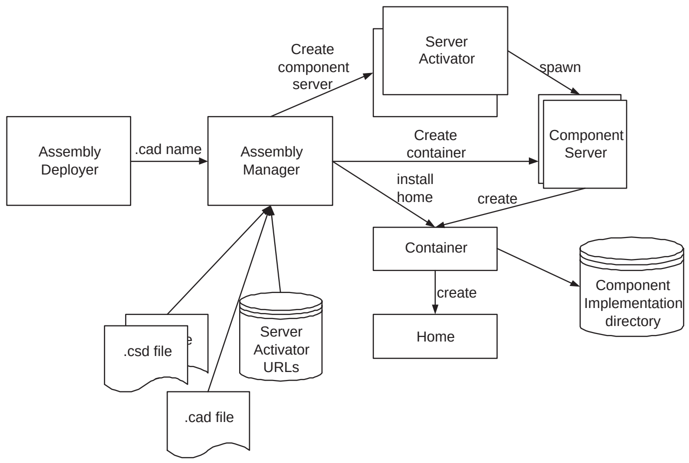

Figure 2: **Online Component Assembly**

Our implementation interprets these .csd and .cad files, and creates and configures the components, their run-time server environments, and QoS properties within the supporting ORB and other related infrastructure. The dynamic version of our CCM implementation currently runs several daemon processes for each deployment environment: one or more Component Installation/Server Activation (CISA) daemons on each machine where components can be deployed, an additional Assembly Manager daemon and an Assembly Deployer process used by the system developer.

The Assembly Manager stores an internal table with the target platform availability information. The Assembly Deployer tells the Assembly Manager which assemblies of components (each assembly is defined in a separate .cad file) should be deployed on which target machines. The Assembly Manager parses the XML structures in the .cad file, and generates its own internal data structure as an intermediate representation of that assembly. The Assembly Manager then traverses this intermediate representation, instructing each CISA daemon to install and configure specific component servers and containers, to create specific homes, and to instantiate specific component instances. Each CISA daemon has additional information about the component implementations available on that endsystem

The dynamic component assembly approach suffers from the following drawbacks:

- XML parsing may be too expensive to be performed during system initialization.
- Multiple process address spaces may be required to coordinate the creation and assembly of components.
- Online loading of component implementation from DLLs or shared objects may not be supported by RTOS platforms like VxWorks where such facilities are not available.

3.2 Static Assembly of Components

To address the drawbacks of dynamic assembly approaches, we implemented an alternative approach to component assembly wherein as much work as possible in the assembly process is done offline. The fundamental intuition in understanding our approach is that in DRE systems the stages of the overall system lifecycle are similar to those in more dynamic conventional component-oriented client-server applications. However, in our *static configuration* approach several phases of the CIAO CCM lifecycle are compressed into the compile-time and system-initialization phases, so that (1) for testing and verification purposes the set of components in an application can be identified and analyzed before run-time, and (2) overheads for run-time operation following initialization are reduced and made more predictable. Furthermore, due to the nuances of the platforms traditionally used for deploying DRE systems, not all features of conventional platforms are available. Our approach therefore avoids certain mechanisms that are either unavailable or too costly in terms of performance.

We follow these intuitions in our approach, taking the existing configuration phases in the online approach described in the previous section and pushing several of them earlier in the configuration lifecycle. We ensure that our approach can be realized in the context of platforms like VxWorks, by refactoring the configuration mechanisms and retargeting them to use only the services available on the target real-time platforms.

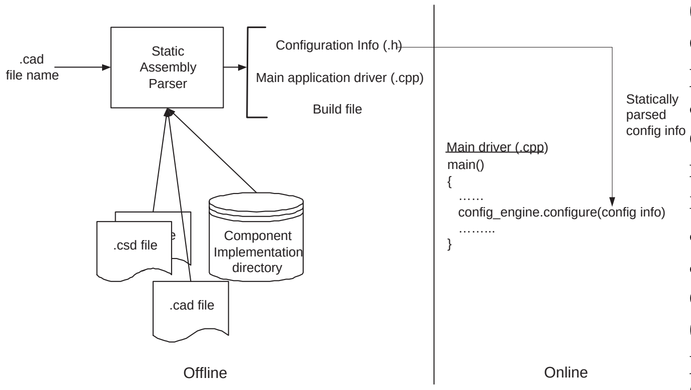

Figure 3: **Static Component Assembly**

In the static configuration approach, shown in Figure 3, configurations XML files are translated in a code generation step just before compile time (managed by the same projectMakefile processes that do the compilation) into C++ header and source files that are then compiled and linked with the main application. The result is that all such XML parsing has been moved off-line (before run-time), and the resulting information is linked statically into the application itself. Each endsystem boots and initializes in a single process address space, so that there is no need for inter-process communication to create and assemble components.

3.3 Component Configuration in CIAO and PRISM

Figure 4 shows the different steps involved in component assembly using CIAO and PRISM. We map the similarities between the individual stages in the two models. The PRISM component model also includes a number of other activities including but not limited to initialization of services like persistence, distribution and concurrency. We focus only on the component assembly part and hence consider initialization of these services out of scope.

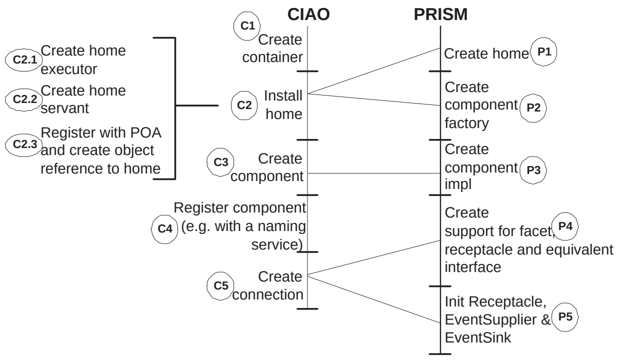
CIAO
PRISM
C1 Create container
C2.1 Create home executor
C2.2 Create home servant
C2.3 Register with POA and create object reference to home
C2 Install home
C3 Create component
C4 Register component (e.g. with a naming service)
C5 Create connection
P1 Create home
P2 Create component factory
P3 Create component impl
P4 Create support for facet, receptacle and equivalent interface
P5 Init Receptacle, EventSupplier & EventSink

Figure 4: **Correspondence between CIAO and PRISM**

The following are the steps in the assembly of components in PRISM. A home object is created (P1) for each component, that is then responsible for creating a component factory (P2) for that component. Each component's factory creates the component implementation (P3). Within the component implementation, the facets, receptacles and equivalent interfaces are created (P4) so that connections can be made from/to other components. Finally the connections between the facets and receptacles are made in (P5). In step (P5), apart from making connections between facets and receptacles, connections are made between event suppliers and event sinks. In PRISM, a connection between an event supplier and an event sink is established by means of the TAO Real-Time Event Channel (RTEC). These correspond to the "publishes" and "consumes" ports in CCM, though it must be noted that currently in the CIAO implementation we do not use the RTEC to connect a publisher and consumer. For our comparisons, hence, we do not take into account the connections established by means of the RTEC. Note that most of the object creations in the above steps are plain C++ objects created on the heap.

The following are the steps in the assembly of components in the CIAO implementation of CCM. A CCM Container object is created (C1) to hold the different components. The container acts as a common place to set different policies such as transaction, persistence, *etc.*. Note that, in contrast, we do not have a corresponding concept of a container in PRISM and hence in our instrumentation we don't consider this for our comparison. A home object is created (C2) and is installed on the container created in (C1). This involves three substeps - A home executor object is created (C2.1) and a home servant object is created (C2.2). The home servant object is then registered with the POA (C2.3) and an object reference created that can then be used to created components using the home. Note that, in contrast with the PRISM component model, there are a lot of CORBA objects being created in the CIAO CCM implementation. The next step (C2.3) is to create components using the home object reference created in the previous step. A component's object reference is advertised with a Naming service (C2.4). This step is an optional step and is done only if it is specified so in the component assembly descriptor XML files. Finally connections are established between matching publisher and consumer ports and facets and receptacles respectively, according to the connection specifications in the descriptor files. The CIAO implementation currently does not use the TAO RTEC to establish connections between publisher and consumer ports. The connection is achieved thru a plain two-way call mechanism.

4 Empirical Evaluation of CIAO

Our objectives in evaluating CIAO's configuration infrastructure are twofold. First, we want to quantify the performance improvement static CIAO offers with respect to initialization time over dynamic CIAO. Second, we want to compare the static configuration implementation in CIAO with the avionics domain-specific PRISM component model [6] used by Boeing.

4.1 Evaluation of Dynamic and Static Configuration Mechanisms

4.1.1 Experiment Design

Experiments to compare the kinds of static and dynamic configuration mechanisms described in Sections 3.1 and 3.2 must be performed on a platform that can (1) support necessary dynamic configuration mechanisms such as shared object libraries, and (2) offer suitable real-time performance. We therefore chose Linux as our experimental platform to evaluate the relative performance of static and dynamic configuration mechanisms in CIAO, as it provides both of these capabilities.

To minimize any effect of varying application logic on our comparisons, we chose to use a very basic scenario within the Boeing Open Experimentation Platform (OEP) for the DARPA PCES program - the Basic Single Processor (BasicSP) scenario. This scenario models the data flow and processing that happens when the information from a navigation GPS device is used to display the route taken by the aircraft on the cockpit display. This scenario uses three components as shown in Figure 12. The GPS device periodically makes current position information available thru its facet interface. It publishes the availability thru its *dataAvailable* port that is connected to the AIRFRAME component. The AIRFRAME component, on receiving the *dataAvailable* "event" from the GPS, gets the position information thru the GPS component facet. The AIRFRAME component, in turn, publishes an event that is received by the DISPLAY component. The DISPLAY component gets the position data from the AIRFRAME and displays the route on the display device. Note that the data flow uses a push-pull model, where an "event" is pushed to indicate that data is ready to be collected. Then those who are interested can collect the data by querying the appropriate component.

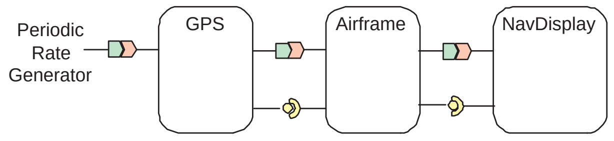

Figure 5: **Boeing BasicSP Scenario**

Experiment testbed. The experiments comparing the dynamic and static configurations of CIAO were performed on a Pentium-IV 2.5GHz with 500MB RAM, 512KB cache and running RedHat Linux 9. For the timing measurements, we used the high resolution timer implementation that internally uses the Pentium Timestamp counter (TSC). The TSC increments every tick of the processor thus offering a nanosecond resolution timestamp. CIAO 0.4.1 was used for the comparison.

4.1.2 Empirical Results

We use histograms to do all the comparisons due to the fact that they offer both insights into the density of distribution of the samples and a visual estimate of the differences. We can also determine the upper and lower bound on outliers in the histogram.

Time for assembly. Figure 13.a shows the total time taken for assembling all the components - this includes creating homes, containers and components and establishing necessary connections between the components. The static config-

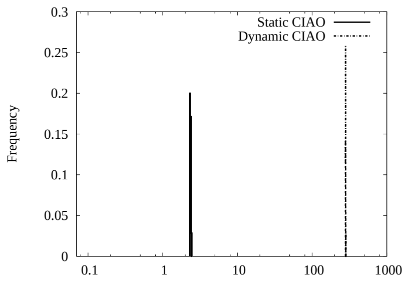

Figure 6: **Static vs Dynamic: Total Time Taken for Assembly**

uration approach takes very little time (avg=2.36msec) when compared to the dynamic approach (avg=281msec). This performance improvement is not very surprising since the dynamic configuration parses XML files at runtime and also loads shared objects containing component implementations, both of these avoided in the static approach. We now proceed to analyze the individual segments to investigate which segment contributes the most to the longer time for the dynamic approach.

Time for component server creation. This comparison is shown in Figure 14. It is obvious that this stage contributes the most (avg=125.41msec) to the delay observed in the dynamic approach. This is the stage in which a component server process is spawned off in the dynamic approach, whereas in the static approach a component server object is created in-process (avg=0.1msec) right at the beginning. Spawning off a separate process incurs a lot of overhead and this is reflected in the dynamic approach plot.

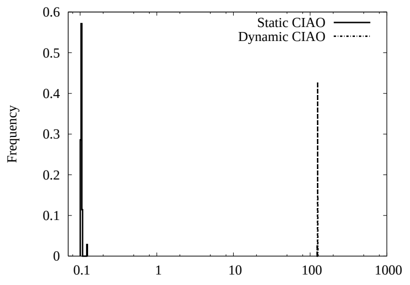

Figure 7: **Static vs Dynamic: Component Server Creation time**

Time for home creation. Figure 15 shows this comparison. This is shared objects/DLLs, which is relatively expensive. The three different lobes for the dynamic configuration could be attributed to the slight differences between components and the XML parsing associated with their home descriptors in the XML file. Here again, the static approach takes very little time (avg=0.15msec) when compared to the dynamic approach(avg=24.16msec).

Time for container/component creation, component registration, and connections. These comparisons are shown in Figure 16, Figure 17, Figure 18 and Figure 19 respectively. There is not much difference between the two in terms of these activities. The slight difference between the two approaches can be attributed to the XML parsing overhead incurred by the dynamic approach. Moreover, the dynamic approach uses the visitor pattern to traverse the parsed XML data structure which incurs a slight overhead (*e.g.*, double dispatching) when compared to the simple for loop construct used by the static approach to traverse the offline-parsed component information stored in tables.

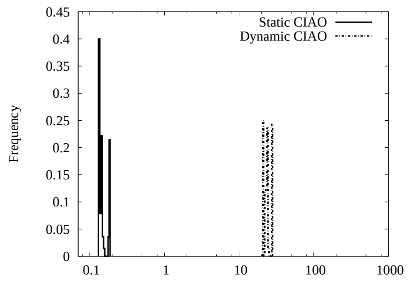

Figure 8: **Static vs Dynamic: Home Creation time**

	Disk Space	Memory Size
Static BasicSP	56,258 K	5,246 K
Dynamic BasicSP	87,127 K	15,323 K

Table 1: **Staic vs. Dynamic CIAO Footprint Summary**

Footprint measurements. Table4.2.2 shows a comparison of the footprints for the BasicSP application using static and dynamic configurations of CIAO.

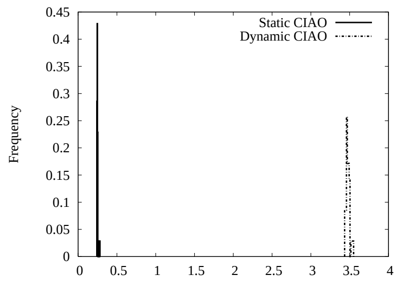

Figure 9: **Static vs Dynamic: Container Creation time**

4.2 Evaluation of Static Configuration in PRISM and CIAO

4.2.1 Experiment Design

Experiments to compare open standards-based component models to domain-specific component models must be run within the platform for which the domain-specific component model implementation was designed. We therefore used a PowerPC board running VxWorks to evaluate the performance of static configuration mechanisms in PRISM and CIAO.

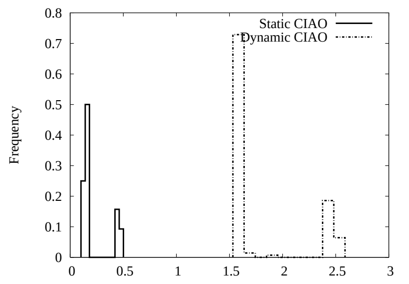

Figure 10: **Static vs Dynamic: Component Creation time**

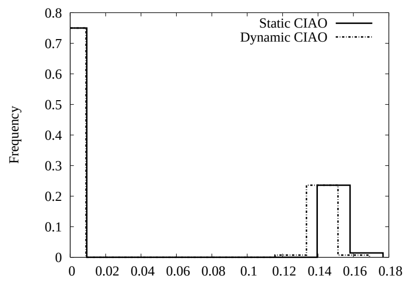

Figure 11: **Static vs Dynamic: Component Registration time**

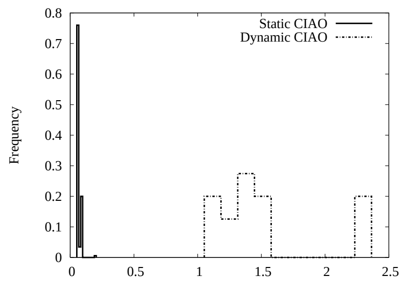

Figure 12: **Static vs Dynamic: Connection Creation time**

We instrumented the relevant parts of the PRISM component model that were provided as part of the Boeing PCES OEP release 3.0. For these experiments we again used the application scenario and components shown in Figure 12 and described in Section 4.2.

Experiment testbed. The experiments comparing PRISM and CIAO static configuration were run on a Motorola 5110-2263 VME board with a MPC7410 500 Mhz processor on a 100Mhz bus with 512MB RAM and running VxWorks. High resolution timestamps were taken at the beginning and end of an interval. A high resolution timestamp consists of a kernel tick counter and a system tick counter. The kernel tick counter, obtained using the VxWorks *tickGet()* call is a low-resolution tick counter that advances every 5msec. The system tick counter, obtained using *sysTimestamp()* call, is a high-resolution tick counter that advances every 40ns and reinitializing to 0 every 5msec. These tick counters can be combined to obtain the time elapsed since system start and their difference gives elapsed interval time, which is then converted to nanoseconds. Boeing PCES OEP release 3.0 was used for the PRISM measurements. These experiments were conducted using a post-1.4 (prerelease) version of ACE/TAO/CIAO.

4.2.2 Empirical Results

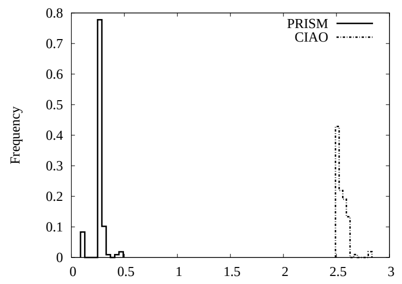

Figure 13: **PRISM vs CIAO: Time for Home Creation**

Time for home creation. Figure 20 shows the time taken by PRISM and CIAO to create a home object. In the case of PRISM, home object is just a native C++ object and hence the home creation time is the time for one dynamic memory allocation and some subsequent initialization of the home object. In the case of CIAO, the home creation is a more elaborate process involving creation of a home executor and home servant. The home servant is registered to the POA and an object reference created for later use to create components. These are CORBA objects and creating and activating them is more expensive than creating a native C++ object. We believe this to be the reason for the difference in home creation between CIAO and PRISM. We believe the smaller lobes for both PRISM and CIAO to be outliers. We believe another key overhead in CIAO could be from the different CORBA related operations, *e.g.*building CORBA policy lists, especially since such operations could possibly involve *CORBA::Any* types.

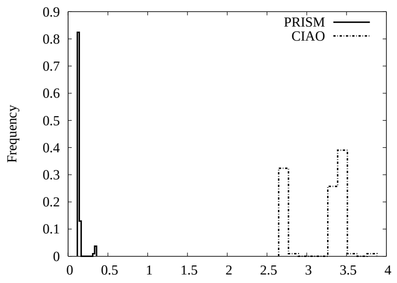

Figure 14: **PRISM vs CIAO: Time for Component Creation**

Time for component creation. Figure 21 shows a comparison of the component creation time in PRISM and CIAO. For the CIAO case, the different lobes correspond to different components - the left lobe corresponding to the DISPLAY component and the right lobe corresponding to the DEVICE and AIRFRAME components. We believe that this variation is due to differences in the generated component initialization code generated for the DISPLAY component vs the other components. As shown in Figure 12, the DISPLAY component has only one receptacle and one consumer port. The timer trigger to the DEVICE component has been implemented using another component which sends periodic timer events to the DEVICE component. Hence the DEVICE and AIRFRAME components have each a facet, a receptacle and a publisher port. We surmise that the two lobes could represent such dissimilarities between components. The PRISM model does not show this pronounced variation because the objects are native C++ objects, as opposed to CORBA objects in CIAO, and the resulting difference is hence very minimal. We believe that the creation of CORBA objects vs the creation of C++ objects is again the contributing factor between the CIAO and PRISM models in terms of creation of components.

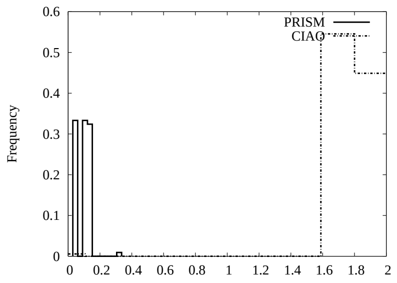

Figure 15: **Time for Connection Creation**

Time for connection creation. Figure 22 shows a comparison of the connection creation time in PRISM and CIAO. We believe that the creation of CORBA objects vs the creation of C++ objects is again the contributing factor between the CIAO and PRISM models in terms of creation of connections also.

5 Related Work

Universal Network Architecture Service (UNAS) [10] is a commercial product that automatically generates software architectures and supports distributed and heterogeneous software systems. One of the main strengths of UNAS is its ability to rapidly build, execute and experimentally compare a set of reasonable architectural design alternatives. While UNAS is a proprietary CASE tool that addresses concerns of large-scale software development, the static configuration approach that we discuss here is an alternative approach to dynamic configuration in the context of our specific CCM implementation and could be chosen as a strategy during code generation as part of a larger CCM tool-chain which conforms to open standards from OMG.

6 Concluding Remarks

The experiments performed in the dissertation show that CIAO adds only a small amount of overhead by comparing the performance of equivalent CIAO application to that based on TAO. The proportion of overhead will diminish even more with the increase of the size of the payload of an operation. CIAO also does not degrade the predictability of applications by adversely affect the jitter. The current implementation of CIAO, however, does demand more secondary storage and primary memory for running CIAO applications. The impact on extra footprint and storage space is expected to lessen as implementation of CIAO evolves and when the dependencies to unused libraries are removed.

This chapter also shows that CIAO's run-time supports for real-time applications impose only a small amount of overhead to the overall performance and does not adversely affect the predictability. Moreover, CIAO's real-time extension has shown to enable the composition of real-time behaviors into an application flexibly and *effectively*. Because developers can now integrate real-time behaviors over the whole application end-to-end, this extension can make developing, maintaining and validating large scale DRE applications easier.

Based on the results obtained in this section, we now offer the following observations and recommendations for developers of complex real-time systems:

Observation. The experiments performed in this chapter are modeled after existing TAO Latency performance tests and the real-time validation tests for TAO's Real-time CORBA implementation [11]. Comparing CIAO's implementations to their TAO-based counterparts, one striking difference is how easy it is to develop and modify CIAO-based tests. Developing TAO test programs requires writing new tests. That is, several specific programs are required to provide different tests of different configurations, as in the case of basic performance tests for TAO with and without RT-ORB. Contrarily, CIAO requires only one application assembly but different configurations were achieved by using different standard tools provided in CIAO, *i.e.*, by changing the deployment environment configuration to use regular component server or real-time component server.

The real-time validation tests provide even more obvious contrast. TAO-based test programs requires elaborated design of test procedures, options, and configuration into the programs themselves and can only be run using the corresponding Perl scripts. In contrast, CIAO tests use only a limited number of simple component implementations running on common run-time environment. Different test configurations can be selected by simply deploying an assembly with different combination of application composition and real-time behaviors. However, CIAO does require significant more storage space and memory.

Recommendation. This observation shows that although CIAO does impose modest performance overhead, application developed using CIAO actually can achieve superior performance much easier than using TAO as CIAO-based applications are much easier to reason, maintain, modify and enhance. CIAO applications can, therefore, be optimized to achieve better performance with lower cost. Further work on CIAO is needed, however, to reduce CIAO's footprint so that CIAO can be applied to application domains where there is stringent memory limitation. It's important to be able to also make the CIAO implementation more robust and be able to compose CIAO internal modules based on application requirements. Reflective middleware techniques, such as dynamicTAO [12], is one such approach worth investigating.

Observation. The development paradigm provided by CCM allows developers to attach relevant metadata in each lifecycle phase. It is important to "bind" the information at proper lifecycle phase to maintain as much as flexibility. Based on the example scenario in our experiment, we draw the following recommendations:
Recommendation.

- **Bind decisions early if possible.** Some information can be ignored after it is checked at a particular lifecycle phase. For example, after the component packaging phase *ensures* that each event sink has a corresponding executor, the application assembly and system deployment phases need not be concerned with that issue. This has the overall benefit of simplifying decisions made later in the system lifecycle.
- **(Re)bind decisions flexibly.** Other information, and the results of previous decisions that have relied on it, cross-cut several phases of the system lifecycle. For example, after the sets of available event source and sink rates are ensured to match along component dependences in the assembly phase, the resulting sets of rates and the component dependences are still needed in the system deployment phase. This allows configuration of concerns that cross-cut the architectural boundaries of component, application, and system, as well as the configuration phase for each of these architectural levels.
- **QoS aspects tend to cross-cut functional boundaries.** Notice especially that QoS information is often refined in subsequent lifecycle phases after it is introduced. Functional information, on the other hand, tends to be more fixed once it is specified in a given phase. This reflects a natural point of difference between CIAO and conventional CCM in which functional information tends to compose in a more object-oriented manner, while the "locality of reference" of QoS decisions tends to be organized around aspect modularity that cross-cuts object and even component boundaries. CIAO is designed with the necessary refinement of QoS aspects in mind, and the understanding that decisions can improve with additional information as long as prior decisions can be kept flexible and revisited as needed. Conventional CCM on the other hand is designed more for functional properties that once specified remain stable for all subsequent composition stages.

References

[1] Object Management Group: CORBA Messaging Specification. Object Management Group. OMG Document orbos/98-05-05 edn. (1998)

[2] Schmidt, D.C., Vinoski, S.: An Overview of the CORBA Messaging Quality of Service Framework. C++ Report **12** (2000)

[3] Subramonian, V., Gill, C.: A Generative Programming Framework for Adaptive Middleware. In: Hawaii International Conference on System Sciences, Software Technology Track, Adaptive and Evolvable Software Systems Minitrack, HICSS 2003, Honolulu, HW, HICSS (2003)

[4] Wang, N., Gill, C.: Improving Real-Time System Configuration via a QoS-aware CORBA Component Model. In: Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003, Honolulu, HW, HICSS (2003)

[5] Sharp, D.C., Roll, W.C.: Model-Based Integration of Reusable Component-Based Avionics System. In: Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003. (2003)

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

[7] Pyarali, I., Schmidt, D.C., Cytron, R.: Techniques for Enhancing Real-time CORBA Quality of Service. IEEE Proceedings Special Issue on Real-time Systems **91** (2003)

[8] Douglas Niehaus, *et al.*: Kansas University Real-Time (KURT) Linux. `www.ittc.ukans.edu/kurt/` (2004)

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

[10] Royce, W., Boehm, B., Druffel, C.: Employing unas technology for software architecture education at the university of southern california. In: Proceedings of the eleventh annual Washington Ada symposium & summer ACM SIGAda meeting on Ada, ACM Press (1994) 113–121

[11] Pyarali, I., Schmidt, D.C., Cytron, R.: Achieving End-to-End Predictability of the TAO Real-time CORBA ORB. In: 8^{th} IEEE Real-Time Technology and Applications Symposium, San Jose, IEEE (2002)

[12] Kon, F., Campbell, R.H.: Supporting Automatic Configuration of Component-Based Distributed Systems. In: Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems, San Diego, CA, USENIX (1999) 175–178