PARIGE: Ensuring Deployment Predictability of Distributed Real-time and Embedded Systems

Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale
Dept. of EECS, Vanderbilt University, Nashville, TN
{dengg,schmidt,gokhale}@dre.vanderbilt.edu

Abstract

Dynamic deployment and configuration (D&C) of components in response to environmental changes or system mission mode changes is essential to facilitate runtime resource management for component-based distributed real-time and embedded (DRE) systems. It is therefore essential that D&C can be performed a timely and predictable manner. This paper provides three contributions to the study of predictable D&C for component-based DRE systems. First, we describe how the predictability of component-based D&C can be affected by application dependency relationships and priorities. Second, we describe how a multi-graph algorithm called partial priority inheritance via graph recomposition *(PARIGE) can improve D&C predictability. Third, we empirically evaluate the effectiveness of PARIGE on a representative DRE system based on NASA Earth Science Enterprise's Magnetospheric Multi-Scale (MMS) mission system. The results show that PARIGE incurs negligible ∼1% D&C performance overhead, but can avoid unbounded deployment time priority inversion when component assemblies with different priorities have complex dependencies among each other, thereby significantly improving the responsiveness of mission-critical tasks with higher priorities.*

Keywords: Component middleware, Distributed Real-time and Embedded systems, Deployment and Configuration.

1. Introduction

Emerging trends and challenges. Developing distributed real-time and embedded (DRE) systems whose quality of service (QoS) can be assured even in the face of changes in available resources or QoS requirements is an important and challenging R&D problem. Systems with such characteristics are called *open* DRE systems [1] since they operate in open environment and must be prepared to accommodate changing operating conditions or requirements, such as power levels, CPU/network bandwidth or mission modes. Examples of open DRE systems include shipboard computing environment [2], and intelligence, surveillance and reconnaissance systems.

Open DRE system are often large and complex, *e.g.*, a shipboard computing system may consist of thousands of software components that run a wide range of missions, such as ship navigation, ship structural health monitoring, vision-based object tracking and object characterization. To manage the overall complexity of such systems, the missions are often decomposed into many domain-related tasks that can be modeled as *operational strings* [4], which are assemblies of software components that capture the partial order and workflow of a set of executing software capabilities for particular domain tasks.

Operational strings can be deployed onto multiple nodes of the target running environment, and different components in operational strings can communicate remotely with each other. Operational strings often run concurrently in the same target environment and share many system resources, such as CPU, memory, and network bandwidth. Typically, to achieve certain mission goals different operational strings can cooperate with each other through their *ports*, which delegate to the ports of monolithic components that consist of the operational strings.

In complex DRE systems, many operational strings may be deployed dynamically, *e.g.*, in response to mission mode or environmental changes. If dependencies exist among these operational strings, *deployment priority inversions* can occur at runtime. A deployment priority inversion occurs when a higher priority operational string cannot be deployed before lower priority operational string(s) because of the dependencies between them. Existing D&C frameworks [5, 6, 7] only consider the dependency between operational strings and ignore their priorities, which can cause unbounded deployment priority inversions for DRE systems.

Solution Approach → Partial Priority Inheritance via Graph Recomposition.

To address the challenges of open DRE systems described above, we developed a technique based on an algorithm called *partial priority inheritance via graph recomposition* (PARIGE). This algorithm analyzes the dependencies between operational strings and removes

all the dependencies from higher priority operational strings to lower priority ones by promoting[1] components from lower priority operational strings to higher priority ones. By applying our technique, a D&C framework can avoid potential priority inversions when multiple operational strings are deployed at runtime.

The three three main steps of our approach are as follows:

- Step 1 converts a deployment descriptor (which contains metadata describing a set of operational strings) into an in-memory *directed graph* representation. Each vertex in the graph represents a component in the operational string and each edge represents a connection between two components.
- Since a deployment plan may have multiple operational strings with different priorities having dependencies among each other, step 2 analyzes the dependency relationship between all the operational strings by performing a graph-based algorithm called *partial priority inheritance via graph recomposition* (PARIGE). This algorithm removes all the priority inverted dependencies between operational strings by promoting component(s) from the lower priority operational string to the higher priority string.
- After graphs are recomposed, step 3 converts them back to deployment descriptor format and fed to the D&C framework for deployment. For the operational strings with dependencies on each other, the D&C framework can then deploy the operational strings from the highest priority to the lowest priority.

When a DRE system has many operational strings with complex dependencies it is hard to determine manually which components in which operational strings should be promoted and which operational string to promote. This paper therefore makes the following three contributions to the research on D&C for component-based DRE systems:

- Analyze dependency relationships among operational strings to determine how each relationship can affect deployment predictability.
- Present a multi-graph algorithm called "partial priority-inheritance via graph recomposition" to avoid deployment priority inversion.
- Empirically evaluate the multi-graph algorithm to determine how effective it is on a representative DRE system.

1 In the context of this paper, *promoting* a component means that before this component is deployed it is temporarily moved from a lower priority operational string to a higher priority operational string for deployment purpose only.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes a representative DRE system case study that elicits key challenges to ensure the predictability of operational string D&C; Section 3 presents the PARIGE algorithm based on a multi-graph recomposition technique that resolves these challenges; Section 4 presents results of experiments that evaluate our techniques empirically; Section 5 compares our work with related research; and Section 6 presents concluding remarks and lessons learned.

2. Motivating Case Study

This section describes different configurations of operational strings in DRE systems that can cause deployment priority inversion to occur due to the dependencies among the strings. To make our discussion concrete, we use NASA's Magnetospheric Multi-Scale (MMS) mission system [8] as a case study. We first present the case study and then identify key challenges that must be addressed to ensure D&C predictability for the case study.

2.1. Overview of NASA MMS Mission System

The NASA Earth Science Enterprise's MMS mission system system uses five satellites with multiple sensors on each satellite to perform solar-terrestrial probe tasks. The satellites orbit the earth in formation, and collect electromagnetic and particle data in the earth's magnetosphere. The MMS mission operates in three data modes: *slow*, *fast*, and *burst*. These data modes may also include different goals, orbits, and data priorities. Each satellite must be capable of determining the necessary task sequences to achieve prescribed goals based on the current environmental and system conditions, as well as revising task sequences in response to changing conditions.

To achieve autonomy, an automated planner is deployed within the MMS system to handle autonomous mode changes driven by the satellite position and the results of analyzing collected data. The task sequences are implemented by components for coordinating the trajectory and orientation of satellites, sensor selection and data collection for individual satellites, and data integration and compression to create telemetry streams that are beamed down to earth stations.

Figure 1 shows three operational strings that a planner generates for a mission task in one of the satellites. Each operational string has different deployment priority (*i.e.*, high, medium, and low) that are determined by how each operational string is accessed by the overall MMS system. The three operational strings are briefly described as follows:

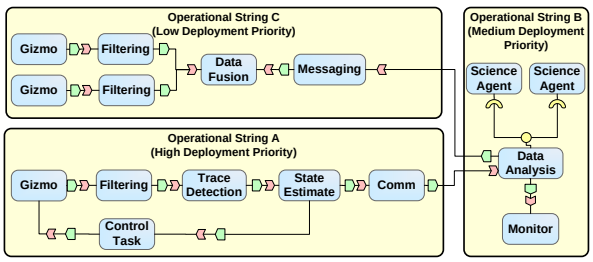

Figure 1: Operational Strings Generated by Planner

- **Operational string A** defines a mission-critical task that collects field data when a satellite moves to particular locations. To ensure this task is performed properly, the operational string must be deployed as fast as possible to avoid loss of data. Operational string A can store the collected data in its own data store, but can also send the data to other operational strings through its event sources.
- **Operational string B** is designed for a domain-centric data analysis. Different scientific analysis tasks can be configured through the facets of components of this operational string. For example, Science Agent components can be configured to achieve scientific objectives, such as analyzing models of complex phenomena like extended weather forecasting.
- **Operational string C** is for less essential data analysis task and can collect auxiliary field data, such as Sun zenith, satellite view zenith [9], which can serve as additional input for analysis. This operational string only operates occasionally, *e.g.*, when the data analysis component in operational string B explicitly issues a request to request such data as additional input for scientific analysis models. The components in operational string C are driven by events exchanged through their event sources and sinks.

Operational strings are organized from domain perspective, *e.g.*, each operational string is designed to accomplish certain domain tasks, such as collecting certain field data, or perform certain analysis on different data models. In our MMS scenario, operational string A services (*e.g.*, collecting essential field data for scientific analysis) are most important for the MMS system, so it has the highest deployment priority among the three operational strings. Conversely, operational string C has the lowest deployment priority among the three since it is designed as a less essential service, *i.e.*, collecting auxiliary data only when necessary. Finally, operational string B is designed to have medium deployment priority because its scientific analysis role is less important than operational string A, but more important than operational string C. Operational string B, however, needs to send events to operational C to notify it to collect auxiliary field data and perform analysis when necessary. The deployment priorities of operational strings are not the same as execution priorities because the latter deals with real-time QoS at run-time.

As shown in Figure 1, there are two dependencies between operational strings: from *A to B* and from *B to C*. These dependencies cross the boundary of an individual operational string. We therefore call them *external dependencies*, in contrast to those dependencies within an operational string.

2.2. Challenges of Ensuring D&C Predictability in the MMS Case Study

Below we describe four challenges that arise when operational strings are deployed dynamically in open DRE systems, such as the NASA MMS mission case study described above.

Challenge 1: Avoid deployment priority inversion between two operational strings. In conventional D&C technologies, such as the OMG D&C specification [10, 6], when a component of an operational string has a connection (either facet/receptacle or event sink/source) to another component in a separate operational string, an *external reference* must be specified to indicate the remote component and the provided port in the other operational string upon which it depends. To deploy this operational string successfully, the external reference endpoint of the other operational string must be activated before the deployment of source operational string can occur. When such a dependency is from a higher priority operational string to a lower priority operational string, however, the low priority operational string must be deployed *before* the high priority operational string can be deployed to avoid deployment failure caused by the dependency, which results in a priority inversion at deployment-time.

For example, in our MMS system case study described in Section 2.1 the dependency from operational string B (medium priority) to operational C (low priority) can cause a deployment priority inversion between operational strings B and C. This dependency requires the deployment of operational string C before operational string B to resolve the dependency. Not all components in operational string C need be deployed to resolve the external dependency between B and C.

Challenge 2: Avoid deployment priority inversion propagation effect. A more general priority inversion situation involves multiple operational strings. In this case, to resolve a dependency from a higher priority string to a lower priority string, not only must the

lower priority string be deployed before the high priority operational string, but also the operational strings the lower priority string depends on. When these operational strings have lower priority than the high priority string, however, deployment priority inversion will occur between operational strings.

For example, in our MMS system case study operational string A has a high priority and an external dependency on operational string B. More specifically, it is the `Data Analysis` component of operational string B that A depends on. The `Data Analysis` component further depends on the `Messaging` component in operational string C, however, which can cause another deployment priority inversion between A and C.

3. An Algorithm for Partial Priority Inheritance via Graph Recomposition

This section describes how we resolved the challenges described in Section 2.2 using an algorithm called *partial priority inheritance via graph recomposition* (PARIGE). This algorithm converts each operational string into a graph, where each vertex and edge of the graph represent a component and a connection/dependency between two components, respectively. If there is an external dependency between operational strings, then the graph converted from one operational string will have a special type of vertex that represents the external dependency. This special vertex type contains information about the actual refereed operational string and the component in the operational string that it depends on.

The PARIGE algorithm promotes components from one graph to another based on operational string characteristics, including their priorities and their dependency relationships with other operational strings in the same deployment request. After graphs for all the operational string are recomposed to account for the component promotion, a new set of operational strings will be populated from these recomposed graphs, which can avoid deployment priority inversion.

3.1. Overview of the PARIGE Algorithm

Although the PARIGE algorithm recomposes operational strings by promoting components from one operational to another, it has also the following properties that makes it well-suited for D&C of DRE systems:

1. The PARIGE algorithm does not affect the functional behavior of component-based DRE systems.

The PARIGE algorithm evaluates the component dependency relationships and their priorities and recomposes these operational strings to avoid deployment priority inversion. From the perspective of *all* operational strings to be deployed, however, the individual monolithic components and their connections among each other are not modified by the algorithm. In particular, the effect of the PARIGE algorithm on operational string recomposition is only visible for the D&C framework, which does not affect the running system's functional behavior. This algorithm thus does not affect the functionally of operational strings because the topology of all the operational strings (including all the monolithic components and connections) that fulfills functional behavior of the system remains unchanged.

2. The PARIGE algorithm does not affect the QoS behavior of operational strings. When components are promoted from a lower priority operational string to a higher priority operational strings, the priority of the components is also bumped up to match the priority of the higher priority string, which is essential for a task to avoid priority inversion at deployment-time [11]. Since the PARIGE algorithm only promotes components that one or more higher priority operational strings have dependencies on at deployment time, it does not change the actual real-time priorities or other real-time QoS configurations designed for run-time. Therefore, the PARIGE algorithm does not affect the QoS behavior of operational strings.

Figure 2 presents an overview of the PARIGE algorithm by showing an example with three operational strings having priorities high, medium, and low. The dotted and solid arrows represent dependencies between operational strings. In particular, the dotted arrows in the figure represent priority inverted dependencies, *i.e.*, dependencies from higher priority operational strings to lower priority operational strings. Likewise, the solid arrows represents external dependencies without causing priority inversion.

The numbered vertices in Figure 2 denote the vertices promoted from one graph into another. For example, in the first iteration of the algorithm, one vertex is promoted from the medium priority operational string to the high priority operational string and another vertex is promoted from the low priority operational string to the high priority string. In the second iteration, another vertex is promoted from the low priority operational string to the medium priority string.

The PARIGE algorithm recomposes the graphs by parsing the input set of graphs and removing dotted arrows by promoting some component(s) from a lower priority operational string to a higher priority string. This process may introduce some new dependencies between operational strings due to component promotion. The algorithm, however, only introduces solid arrows, *i.e.*, only dependencies from lower priority operational

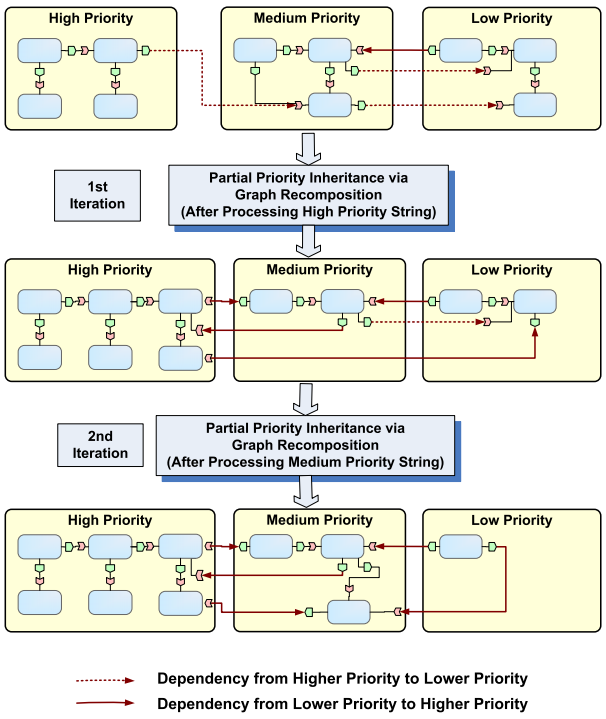

Figure 2: PARIGE Algorithm in Action

strings to higher priority strings exist after the recomposition.

When the algorithm finishes, all dotted arrows in the graphs will be removed and there will be no dependencies from higher priority operational strings to lower priority operational strings. As a result, both priority inversions at run-time and deployment-time can be avoided.

3.2. Design of The PARIGE Algorithm

The goal of the PARIGE algorithm is to remove *all* dependencies from higher priority operational strings to lower priority operational strings. To accomplish this, the algorithm starts with the operational string having the highest priority and processes all the external dependencies of this operational string. After all external dependencies from the highest priority operational string are removed, the algorithm then processes the operational string with the next highest priority. When multiple operational strings have the same priority, we apply the following tie-breaking policies sequentially: (1) evaluating the second metric of each operational string, if given, (2) evaluating the number of external dependencies to the same priority operational strings and treating the operational string with the least number of external dependencies as the higher priority than others, and (3) breaking the tie randomly if such a tie still exists.

When processing an external dependency from a higher priority operational string to a lower priority operational string, the algorithm must trace the dependency into other operational strings and promote components from them if the lower priority operational string has dependencies to them. For example, if a high priority operational string depends on a component X in a medium priority operational string, and if component X also has dependency on a component in a low priority operational string, then the component in the low priority string also must be promoted into the high priority string.

We define a *dependency trace* as a totally ordered sequence S. Each element in the sequence is a component of an operational string that has a priority value associated with it. The starting element of the sequence is the source component of the external dependency of interest. The PARIGE algorithm analyzes all the dependency traces in the operational strings and recomposes the operational strings based on dependency trace characteristics.

3.2.1. Promotion of Components Between Two Operational Strings In this case, a dependency occurs between two operational strings, where a high priority operational string has a dependency on a lower priority operational string.

The unique characteristic of this category is that the dependency trace does not cross the boundary of the lower priority operational string. Since no other operational strings are involved besides the two operational strings of interest, removing such a priority inverted external dependency only requires promoting all components in the dependency trace from the lower priority operational string to the high priority one.

3.2.2. Promotion of Components Across Multiple Operational Strings This more general case involves multiple operational strings, with a dependency trace that spans across the operational strings. A dependency trace that spans across multiple operational strings can be further categorized into the following two situations.

1. Ordered dependency trace. Figure 3 shows an ordered dependency trace. In an ordered dependency trace the priorities of each element in the sequence have a non-increasing order, *i.e.*, all external dependencies in the sequence are priority-inverted. As a result, all the priority-inverted external dependencies must be removed through the component promotion mechanism described in Section 3.1. The category described in Section 3.2.1 where only two operational strings are involved is a special case of an ordered de-

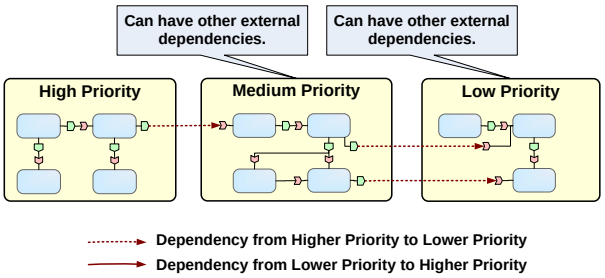

Figure 3: An Ordered Dependency Trace

pendency trace. To remove all priority-inverted external dependencies, the PARIGE algorithm simply promotes all components in the dependency trace into the operational string where the first component of the dependency trace is located.

2. Unordered dependency trace. Figure 4 shows an unordered dependency trace, where the priorities of the elements in the dependency trace do not have a particular order, *i.e.*, some external dependencies are priority-inverted (shown as dotted lines), whereas others are not (shown as solid lines). The PARIGE al-

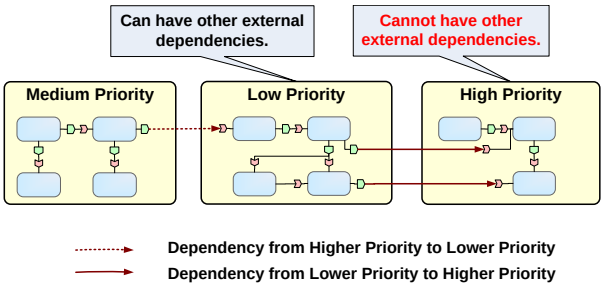

Figure 4: An Unordered Dependency Trace

gorithm always starts with the highest priority operational string and removes all external dependencies on it before moving to the next operational string. The algorithm therefore ensures that in an unordered dependency trace, the elements whose priorities are higher than that of the starting element will have no external dependencies, which ensures the convergence of the algorithm.

For example, given the three operational strings from Section 2, if the high priority operational string has an external dependency on a component in low priority operational string and this component must be promoted into the medium priority operational string. When this promotion happens, the high priority string will depend on the medium priority string, which introduces additional priority-inverted external dependencies.

To remove all priority-inverted external dependencies of an unordered dependency trace, we break the entire dependency trace into two concatenated segments. As shown in Figure 5, the first segment is a priority unordered subsequence, where all the priorities of operational strings are lower than the priority of the source of the dependency trace. The second segment is a prior-

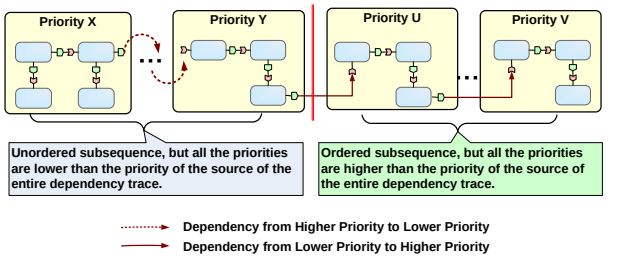

Figure 5: Two Partitions of An Unordered Dependency Trace

ity ordered subsequence, where all priorities are higher than the priority of the source of the dependency trace. For the first segment, we can promote all the components in the subsequence into the operational string where the first component of the dependency trace is located, which will ultimately result in an ordered dependency trace.

The PARIGE algorithm uses multi-graph breadth first search (BFS) to trace dependencies and graph reconstruction to promote components and connections between components. Due to the paper space limitation, we do not include detailed design of the algorithm in this paper. For a reference to the algorithm design, please refer to our technical

3.3. Analysis of the Algorithm

To show that it is possible to apply the PARIGE algorithm at run-time to deploy operational strings dynamically, we now analyze the effects of the PARIGE algorithm for actual operational string deployment.

3.3.1. PARIGE Algorithm Effects on Operational String Deployment Two effects that the PARIGE algorithm could have on the predictability of operational string deployment are described below.

Operational string growth effect. This effect measures the cost of promoting a number of components from lower priority operational strings to higher priority operational strings. Since the deployment of each component takes time and consumes resources,

the fewer components that are promoted, the more benefits the algorithm can provide since priority-inverted dependencies can be satisfied without deploying many components in lower priority operational strings. In the worst case, *all* components from lower priority operational strings could be promoted to higher priority operational strings, which essentially merges different operational strings together. In production DRE systems, such worst cases happen rarely, *i.e.*, all the components in all operational strings have just only one dependence trace. Even in such situations, the PARIGE algorithm still performs the same as a conventional approach that does not take priority into account and only accounts for dependencies among operational strings.

Component host distribution effect. This effect means that due to the promotion of components, components that can be deployed by contacting the `NodeManager` once now contacts the same `NodeManager` multiple times during deployment. Such an effect can increase the overall deployment time due to the increasing number of round trip delays. One way to alleviate this problem is to increase the parallelism among different nodes by using asynchronous techniques between the `ExecutionManager` and `NodeManagers`, such as the Asynchronous Method Invocation (AMI) messaging policy provided by CORBA [12]. For example, AMI can coordinate all the `NodeManagers` in the domain parallelism deployment can be achieved among all the nodes, therefore alleviating the component host distribution effects.

4. Empirical Results

To evaluate the benefits of our PARIGE algorithm, we applied it to a representative DRE system prototype of the NASA MMS mission system described in Section 2. This section first describes the characteristics of the hardware and software testbed and explains our experiment design. We then empirically evaluate the effectiveness of our PARIGE algorithm and its performance overhead.

4.1. Experiment Testbed

We used the ISISlab open testbed (`www.dre.vanderbilt.edu/ISISlab`) for all our experiments. Our experiments used up to 6 nodes running Linux FC4 with Ingo Molnar's real-time kernel patches. When operational strings are deployed we use one node to run the central coordinator `ExecutionManager` and the rest of the nodes as the deployment targets.

The NASA MMS mission system prototyped used for our experiments was developed using the CIAO [13] and DAnCE [7] component middleware. This application consists of 45 components grouped together into 3 operational strings.

4.2. Deployment Latency vs. Deployment Priority

Hypothesis. The hypothesis of this experiment is that the PARIGE algorithm can avoid priority inversion when deploying multiple operational strings where higher priority operational strings have dependencies on lower priority operational strings.

Experimental design. We conducted two experiments on different configurations of operational string dependencies. Our first experiment consisted of `3` operational strings, each of which having `15` components evenly distributed into `5` nodes. Therefore, each node has `9` components. The high priority operational string has one dependency on the medium priority operational string, which in turn has one dependency on the low priority operational string. The dependency between two operational strings has low growth rate, *i.e.*, only one component in a lower priority operational string needs to be promoted. Next we conducted another experiment with two operational strings but with high growth rate, *i.e.*, all the component in the lower priority operational string must to be promoted to the higher priority operational string.

Empirical results and analysis.

Figures 6 and 7 shows the end-to-end latency of D&C request for each operational string in the two experiments described above. As shown in the Figure 6,

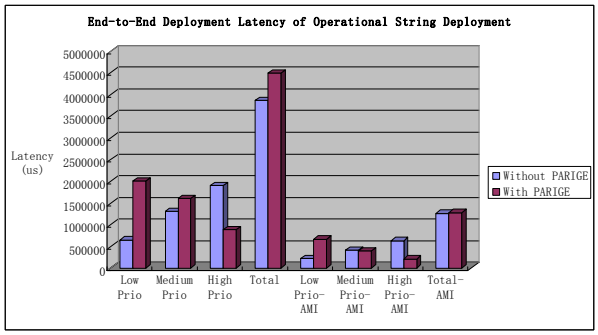

Figure 6: D&C Latency Changes by the PARIGE Algorithm

without applying the PARIGE algorithm, the high priority operational string yields the highest latency while

the low priority operational string yields the lowest latency, while the latency of medium priority operational string lies in between.

In our experiments, there is one dependency from high priority operational string to medium priority operational string and another dependency from medium priority operational string to low priority operational string. Without applying the PARIGE algorithm, therefore, the low priority operational string must be deployed first among the three, followed by medium priority and high priority operational strings, respectively. The PARIGE algorithm removes the priority inverted dependencies which avoids deployment priority inversion, as illustrated in the figure.

Figures 6 also shows how the *component host distribution effect* introduced by the PARIGE algorithm is masked by applying AMI messaging policy, as described in Section 3.3.1. In our experiment, applying AMI improves the performance of the deployment in two aspects. First, the deployment latency of *each* operational string is reduced because of the `ExecutionManager` can coordinate the `NodeManagers` to do deployment in parallel. Second, it masks the component host distribution effect, which results in a reduced total latency of all operational strings, as shown in the figure.

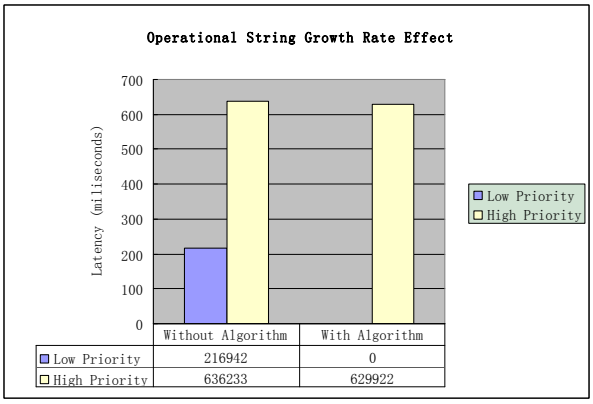

Figure 7: D&C Latency Changes by the Algorithm

Figure 7 shows that after applying the PARIGE algorithm the high priority operational string has the lowest latency since it has no external dependency on any other operational strings. The size change of each operational string is also minimal since the number of promoted components is small due to the dependency trace characteristics.

On the other hand, the dependency between the two operational strings in our second experiment caused all components from the low priority operational string to be promoted to the high priority string, essentially merging the two operational strings together. As a result, the latency of deploying the high priority operational string is nearly the same as deploying it without applying the PARIGE algorithm. However, in a DRE system with multiple operational strings to deploy, it is rare that all components have only one dependence trace, as described in Section 3.3.1.

4.3. Performance Overhead of the PARIGE Algorithm

Hypothesis. The hypothesis of this experiment is that the performance overhead of the PARIGE algorithm is small enough so it can be applied to deploy operational strings at run-time. In contrast to off-line analysis techniques, the PARIGE algorithm must be deployed by `ExecutionManager` to handle requests at runtime, therefore, the PARIGE algorithm should not incur excessive performance overhead to the end-to-end latency of deployment of operational strings.

Experimental design. The experiments consist of 3 operational strings each having 15 components and 2 external dependencies in total. The high priority operational string has one dependency on the medium priority operational string, which in turn has one dependency on the low priority operational string. We first measured the end-to-end latency for deploying all the operational strings without applying the PARIGE algorithm. We then measured the end-to-end latency for deploying increasing number of operational strings with the PARIGE algorithm to measure how much latency overhead was contributed by running the algorithm.

Empirical results and analysis. We first measure the PARIGE algorithm performance itself to determine how its performance is affected by the size of the problem, *i.e.*, number of components (determined by number of operational strings) and number of priority-inverted external dependencies. We then measure its performance overhead against an actual example with 3 operational strings and 2 external dependencies, as described above.

Figure 8 shows the performance result of PARIGE algorithm itself with increasing number of components and number of external dependencies. The results show that the performance of PARIGE algorithm is roughly linear to both the number of components and number of external dependencies. The linear runtime performance characteristics of PARIGE algorithm makes it suitable for dynamically deploying operational strings

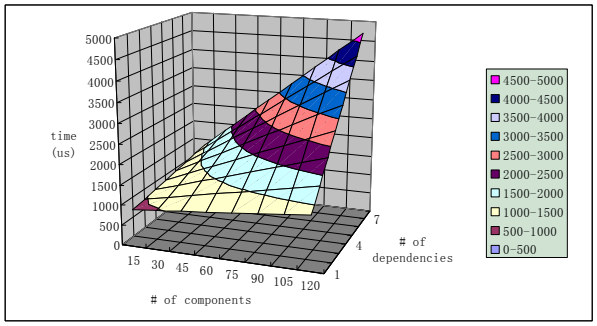

Figure 8: Running Time of the PARIGE Algorithm

online at runtime because the deployment latency of all operational strings exhibits a *linear* time complexity to the number of components in the operational strings.

As long as the performance overhead of the PARIGE algorithm is acceptable to deploy one component, therefore, it should be acceptable to deploy any number of components. To validate this claim, we conducted an experiment that deployed up to 64 operational strings with `960` total components. The results in Figure 9 shows that the deployment latency of all operational strings with and without the PARIGE algorithm. The experiment measures different number of operational strings and different number of components, ranging from `1` operational string with `15` components to `64` operational strings with `960` components. These results show that the actual per-

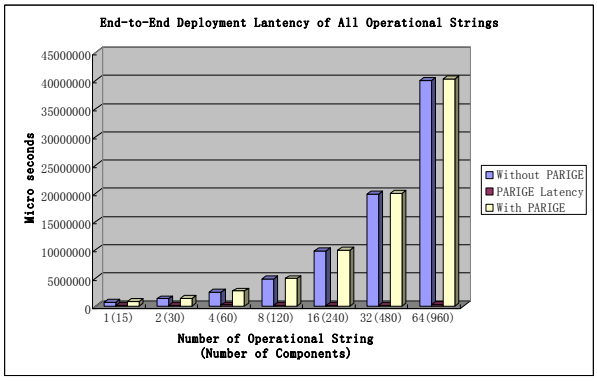

Figure 9: Performance Overhead of the PARIGE Algorithm

formance overhead of PARIGE algorithm for our experiment is consistently less than ∼1%, which further validates our earlier analysis.

5. Related Work

The PARIGE algorithm has many similarities to the priority inheritance protocol (PIP) [14] used for synchronization in real-time systems. The PIP ensures that when a thread blocks one or more high priority threads, it executes its critical section at the highest priority level of all the threads it blocks, *i.e.*, it inherits the highest threads priority. After executing its critical section, the thread returns to its original priority level.

In the PARIGE algorithm, lower priority operational strings are promoted to execute at the priority of higher priority operational strings to avoid deployment priority inversions. The "critical section" in the PIP is thus similar to the "deployment and configuration" activities in the PARIGE algorithm. Our work on the PARIGE algorithm, however, differs from the PIP in the following ways:

- The PARIGE algorithm avoid deadlocks because (1) it removes all priority inverted dependencies between operational strings and then deploy operational strings from the highest priority to the lowest priority sequentially, and (2) it recompose operational strings so circular dependency trace does not cross the boundary of operational strings. In contrast, the PIP may incur deadlocks because of nested resource locks.
- Only part of the operational string is affected, *i.e.*, the PARIGE algorithm just increases the deployment priorities of components with dependencies from higher priority operational strings. In contrast, the PIP does not have such fine-grained level of control because it is a general-purpose scheduling mechanism for resource sharing.
- The PARIGE algorithm is more sophisticated than the priority inheritance protocol because it traverses multiple graphs to identify which components require promotion. In contrast, the PIP is much simpler since it is locality-constrained, *i.e.*, it applies only to one resource and does not concern about how other resources are scheduled.

6. Concluding Remarks

The predictability and scalability of D&C frameworks is essential to support the QoS requirements of open DRE systems. This paper describes a multi-graph algorithm that helps ensure the predictability of deploying multiple operational strings. We first analyze how deployment priority inversion can occur when operational strings have various dependency relationships. We then empirically show how the *partial priority*

inheritance via graph recomposition (PARIGE) algorithm can effectively avoid deployment priority inversions and thus improve the predictability of component deployment in DRE systems.

The following summarizes our lessons learned thus far from developing and applying the PARIGE algorithm to ensure the predictability of deployment of operational strings in DRE systems:

1. The overlap of deployment-time with runtime makes D&C frameworks essential to ensure system QoS. The benefits provided by component middleware significantly change the lifecycle of DRE system development. Due to the complexities of open DRE systems, D&C frameworks assume more responsibilities to ensure system QoS because deployment of system services/components occurs throughout the lifecycle of the systems. By using information available at deployment time, D&C frameworks can effectively identify the complex dependency relationships among operational strings and perform various on-line optimizations, such as the operational string recomposition presented in Section 3.

2. Automated Recomposition of operational strings can help ensure deployment predictability of DRE systems. Although operational strings can simplify the design of DRE systems, it is hard to manually ensure deployment predictability of all operational strings due to the complex dependencies among many operational strings. The PARIGE algorithm presented in this paper enhances the deployment predictability of different operational strings by recomposing operational strings automatically based on the input to the D&C framework and transparently to system deployers.

The PARIGE algorithm is an integral part of DAnCE. Both DAnCE and CIAO are open-source and available for download at `www.dre.vanderbilt.edu/ciao`.

References

[1] Gill, C., Gossett, J., Loyall, J., Schmidt, D., Corman, D., Schantz, R., Atighetchi, M.: Integrated Adaptive QoS Management in Middleware: A Case Study. In: Proceedings of the 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), Toronto, Canada, IEEE (May 2004)

[2] Schmidt, D.C., Schantz, R., Masters, M., Cross, J., Sharp, D., DiPalma, L.: Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. CrossTalk - The Journal of Defense Software Engineering (November 2001)

[3] Sharma, P., Loyall, J., Heineman, G., Schantz, R., Shapiro, R., Duzan, G.: Component-Based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems. In: Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04), Agia Napa, Cyprus (October 2004)

[4] Lardieri, P., Balasubramanian, J., Schmidt, D.C., Thaker, G., Gokhale, A., Damiano, T.: 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 **80**(7) (July 2007) 984–996

[5] Desertot, M., Cervantes, H., Donsez, D.: Frogi: Fractal components deployment over osgi. In: Software Composition. (2006) 275–290

[6] Quéma, V., Balter, R., Bellissard, L., Féliot, D., Freyssinet, A., Lacourte, S.: Asynchronous, hierarchical, and scalable deployment of component-based applications. In: Component Deployment, Second International Working Conference, CD 2004, Edinburgh, UK, May 20-21, 2004, Proceedings. Volume 3083 of Lecture Notes in Computer Science., Springer (2004) 50–64

[7] Deng, G., Balasubramanian, J., Otte, W., Schmidt, D.C., Gokhale, A.: DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In: Proc. of the 3rd Working Conf. on Component Deployment (CD 2005), Grenoble, France, Springer-Verlag (November 2005) 67–82

[8] Suri, D., Howell, A., Schmidt, D.C., Biswas, G., Kinnebrew, J., Otte, W., Shankaran, N.: A Multi-agent Architecture for Smart Sensing in the NASA Sensor Web. In: Proceedings of the 2007 IEEE Aerospace Conference, Big Sky, Montana (March 2007)

[9] Rigole, P., Clerckx, T., Berbers, Y., Coninx, K.: Possibilities to improve ground-based cloud cover observations using satellite application facility (safnwc) products. GEOGRAFIJA. **43**(1) (2007) 21–29

[10] Object Management Group: Deployment and Configuration Adopted Submission. OMG Document mars/03-05-08 edn. (July 2003)

[11] Bettati, R., Liu, J.W.S.: End-to-end scheduling to meet deadlines in distributed systems. In: International Conference on Distributed Computing Systems. (1992) 452–459

[12] Schmidt, D.C., Vinoski, S.: Programming Asynchronous Method Invocations with CORBA Messaging. C++ Report **11**(2) (February 1999)

[13] Wang, N., Gill, C., Schmidt, D.C., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04). Volume 3291., Agia Napa, Cyprus, Springer-Verlag (October 2004) 1520–1537

[14] Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-time Synchronization. IEEE Transactions on Computers **39**(9) (September 1990)