Towards Improving End-to-End Performance of Distributed Real-time and Embedded Systems Using Baseline Profiles

James H. Hill and Aniruddha Gokhale

Vanderbilt University
Nashville, TN, USA
Email: {j.hill, a.gokhale}@vanderbilt.edu

Summary. Component-based distributed real-time and embedded (DRE) systems must be properly deployed and configured to realize an operational system that meets its functional and quality-of-service (QoS) needs. Different deployments and configurations often impact systemic QoS concerns, such as end-to-end response time. Traditional techniques for understanding systemic QoS rely on complex analytical and simulation models, however, such techniques provide performance assurance at design-time only. Moveover, they do not take into account the complete operating environment, which greatly influences systemic performance.

This paper presents a simple technique for searching for deployments that improve performance along a single dimension of QoS concern, such as response time. Our technique uses baseline profiles and experimental observations to recommend new deployments. The results indicate that we are able to use baseline profiles to converge towards solutions that improve systemic performance.

Key words: baseline testing, component-based development, deployment and configuration, performance analysis

1 Introduction

Challenges of developing component-based systems. Component-based technologies are raising the level of abstraction so software system developers of distributed real-time and embedded (DRE) systems can focus more on the application's "business-logic". Moreover, component-based technologies are separating many concerns, such as deployment (*i.e.*, placement of component on host) and configuration (*i.e.*, setting of component properties) [1], management of the application's lifecycle [2], and management of execution environment's quality-of-service (QoS) policies [2, 12], of the application. The result of separating the concerns is the ability to fully address each concern independently of the application's "business-logic" (*i.e.*, its implementation). The technologies that are making this possible include, but are not limited to: CORBA Component Model (CCM), J2EE, and Microsoft.NET.

Although component-based technologies are separating many concerns of DRE system development, realizing systemic (*system wide*) QoS properties, which is a key characteristic

of DRE systems, is being pushed into the realized system's deployment and configuration (D&C) [1] solution space. For example, it is hard for DRE system developers to fully understand systemic QoS properties, such as end-to-end response time of system execution paths until the system has been properly deployed and configured in its target environment. Only then would a DRE system developer determine if different components designed to collaborate with each other perform better when located on different hosts compared to collocating them on the same host, which in turn might give rise to unforeseen software contentions, such as waiting for a thread to handle an event, or event/lock synchronization, or many known software design anti-patterns [13] in software performance engineering [8].

Outside of brute force trial and error, traditional techniques, such as complex analytical and mathematical models [5, 8, 15], can be used to locate candidate D&Cs that meet the required systemic QoS properties. Although such techniques can locate candidate D&Cs, they provide design-time assurance and do not take into account the complete operating environment. Moreover, using these techniques are beyond the knowledge domain of component-based system developers. Component-based DRE system developers, therefore, need simpler techniques that are intuitive and easier to use that can provide assistance in understanding systemic QoS properties in relation to the realized system's D&C.

Solution Approach → Use baseline profiles to understand the deployment solution space. Each individual component in a component-based DRE system has a baseline profile that captures its performance properties when deployed in isolation to other components. As components are collocated (*i.e.*, placed on the same host) with other components, their actual performance may deviate from its baseline performance profile due to unforeseen software contentions. Ideally, to improve systemic QoS properties for a component-based system, system developers should evaluate each individual component's performance against its baseline profile. This will allow them to understand if a component is performing as expected. Moreover, it will allow them to pinpoint possible problems and recommend new deployments that could possibly improve systemic QoS properties.

This paper describes our solution approach for understanding systemic QoS properties, such as end-to-end response time of system execution paths, in relation to the deployment solution space of component-based DRE systems. Our solution uses baseline profiles for each component, which are obtained by instrumenting and profiling the component in a controlled environment. We then execute a deployment of the system in the target environment and compare its performance against the baseline profile. Finally, based on the percentage error between the current test and the baseline profile, and the history of previous deployments, we recommend a new deployment for experimentation that will either improve/degrade system QoS properties. Our solution is designed so that developers not only converge towards a D&C that ensures systemic QoS properties, but allows system developers to get a better understanding of how different deployments influence systemic QoS properties.

Paper Organization. The remainder of this paper is organized as follows: Section 2 introduces a case study to highlight challenges of software contention; Section 3 discusses our technique for understanding the deployment solution space; Section 4 discusses results of applying our technique to our case study; Section 5 presents related work; and Section 6 provides concluding remarks and future research.

2 Manifestation of Software Contention: A Case Study Perspective

In this section we use a representative case study from the shipboard computing environment domain to highlight how different deployment and configurations can introduce different software contention problems that impact systemic QoS properties.

2.1 The SLICE Shipboard Computing Scenario

The SLICE scenario is a representative example of a shipboard computing environment application. We have used this example in prior work and multiple studies, such as the evaluation of system execution modeling tools [4] and the formal verification [3]. In this paper we use it to highlight software contention issues arising out of different deployment and configurations. We briefly reiterate its description here.

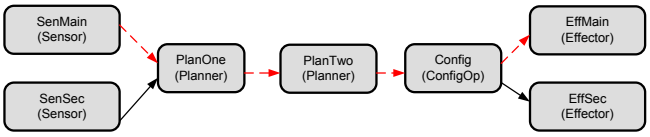

Fig. 1. High-level structural composition of the SLICE scenario.

The SLICE scenario, which is illustrated in Figure 1, consists of 7 different component implementations, (*i.e.,* the rectangular objects): *SenMain*, *SenSec*, *PlanOne*, *PlanTwo*, *Config*, *EffMain*, and *EffSec*. The directed lines between the components represent connections for inter-component communication, such as input/output events. The components in the SLICE scenario are deployed across 3 computing nodes and `SenMain` and `EffMain` are deployed on separate nodes to reflect the placement of physical equipment in the production shipboard environment. Finally, events that propagate along the path marked by the dashed (red) lines in Figure 1 represent an execution (or critical) path that must have optimal performance.

2.2 Software Contention in the SLICE Scenario

We define software contentions as the deviation of performance as a result of the system's implementation—also characterized as performance anti-patterns [13]. In many cases hardware contention does not have noticeable affects on resource utilization. For example, it is possible to have high end-to-end response time and low CPU utilization if different portions of the software components (*e.g.*, threads of execution) unnecessarily hold mutexes that prevent other portions of the software (or components) from executing. The (software) component holding the mutex may not be executing on the CPU, however, it can be involved in other activities, such as sending data across the network. Likewise, it is possible to have high response time if too many components are sending event requests to a single port of another component [10], which increases the queuing time for handling events.

Figure 2 illustrates the results from prior work [4] that used brute force ad-hoc techiques to improve critical path end-to-end response time for the SLICE scenario. As shown in Figure 2, each test respresents a different deployment and each deployment yields a different

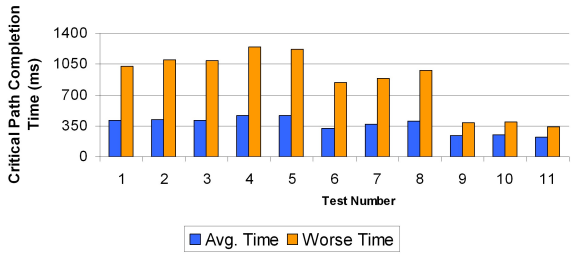

Fig. 2. Prior results of SLICE scenario.

critical path end-to-end response time. For example, tests 10 and 11 have a critical path end-to-end response time difference of $\sim$ 30 msec. We believe this is due to unforeseen software contention resulting from collocating components that were designed and implemented in isolation to other components. The remainder of this paper, therefore, details how we use baseline profiles to understand the deployment solution space and improve systemic QoS performance due to unforeseen software contentions using the SLICE scenario as an example case study.

3 Using Baseline Profiles to Understand the Deployment Solution Space

This section discusses our technique for using baseline profiles to explore and understand the deployment solution space of component-based DRE systems.

3.1 Representing Deployment Solution Space as a Graph

After system composition, a component-based system becomes operational after its deployment and configuration in the target environment. There are, however, many ways to deploy a component-based systems, such as the SLICE scenario in Section 2. For example, Table 1 shows two unique deployments for the SLICE scenario where both differ by the placement of one component; the *SenMain* component on either Host1 or Host2.

Table 1. Example of unique deployments in the SLICE scenario.

	Deployment	
Host	A	B
1	SenMain, SenSec Config	SenSec, Config
2	EffSec, PlanOne	EffSec, PlanOne SenMain
3	PlanTwo, EffMain	PlanTwo, EffMain

When we consider all the unique deployments in the deployment solution space for a component-based systems, it is possible to represent such a space as a graph $G = (D, E)$ where:

- D a set of vertices $d \in D$ in graph G that represents a unique deployment d in the deployment solution space D. For example, deploying *SenMain* then *SenSec* on a single host H_i is the same as deploying *SenSec* then *SenMain* on the H_i. This is different from traditional bin packing, which is based on permutations [9].
- E is a set of edges $e \in E$ where $e_{i,j}$ is an edge between two unique deployments $i, j \in D$. Each edge e signifies moving a single component from one host to another host in the system.

Definition 1. *If $C_{h,i}$ is the set of components deployed on host h in system H for deployment i, then a unique deployment is defined as:*

$$\exists h \in H : C_{h,i} \neq C_{h,j}; i, j \in D \wedge i \neq j \tag{1}$$

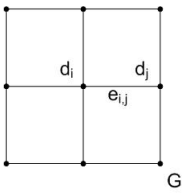

Fig. 3. Simple graph of component-based system deployment solution space.

The resultant graph G, as shown in Figure 3, allows us to create a visual representation of the deployment solution space. The visualization, however, has no concrete meaning from the QoS perspective because developers cannot understand how one deployment differs from another deployment except for the unique combination of components on host. We address this problem by assigning a value to each edge e in graph G using Equation 2,

$$val(e) = \delta \tag{2}$$

which gives rise to a directed graph G'. Figure 4 illustrates G' where δ represents some difference of performance in a single QoS dimension, such as end-to-end response time or system throughput.

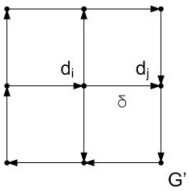

Fig. 4. Directed graph of component-based system deployment solution space.

QoS, however, is known to comprise a N-dimensional space [11], and G' shown in Figure 4 is the deployment solution space for a single dimension of QoS, *i.e.*, visualized on a

single plane. When we consider N dimensions of QoS, we create an N-planar graph G''. Figure 5 illustrates G'' where each plane represents a single QoS dimension. The value of each edge between unique deployments in different planes, *i.e.*, QoS dimensions, forms a N-tuple vector $(\delta_1, \delta_2, ..., \delta_n)$ where $\delta_i = val(e \in G'_i)$ and $i \in$ QoS dimension, $G'_i \in G''$.

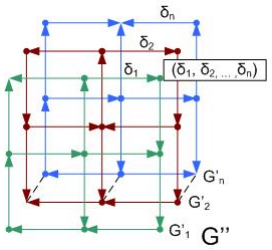

Fig. 5. N-planar directed graph of component-based system deployment solution space for N QoS dimensions.

When also considering the N-planer graph of the N QoS dimension solution space, an edge between the same deployment for each QoS dimension is governed by Property 1. This results in a N-tuple vector equal to zero and is, therefore, ignored.

Property 1. The value of an edge between the same unique deployment in different planes is zero.

Due to the complexity of the N-dimensional QoS solution space in relation to component-based system deployment, we focus on evaluating QoS in a single dimension in this paper. We, therefore, narrow our focus to a 1-planar directed graph (Figure 4), which drastically reduces our solution space. The following section details how we can search this solution space to understand and locate deployments to improve systemic QoS properties along a single dimension.

3.2 Using Baseline Profiles to Search Deployment Solution Space

Each component in the component-based system produces a baseline profile that details its optimal performance. Such performance is realized when the component is deployed into a controlled and isolated environment, *i.e.*, no interference from other components in the system. As a component is collocated with other components, its performance will begin to deviate from its baseline performance. This is due to both software and hardware contentions.

Hardware contentions can be resolved using traditional analysis techniques, such as queuing theory [8], because they have more deterministic properties, such as service rate/time, that can be captured in a mathematical model. Software contentions, however, require execution in the target environment to fully locate and understand because it is hard to capture such problems using a model.

In Section 3.1, we discussed how the deployment solution space in a single QoS dimension could be represented as a directed graph G'. Each directed edge $e \in G'$ represents the difference in QoS performance between two unique deployments. When trying to locate candidate D&Cs that meet a specific QoS performance, or improve systemic QoS performance, it is ideal to

locate deployments that yield performance metrics for each individual component that are close to their baseline. The problem becomes hard when all components and their interactions must be taken into account all at once.

We, therefore, use the percentage error (Equation 3) between observed and baseline performance of a component and deployment history to search the deployment solution space. We do not use the absolute value in the numerator when calculating the percentage error because the positive/negative sign signifies if QoS improved/worsened. Algorithm 1 lists our algorithm for locating a new deployment after labeling the current edge with the percentage difference in QoS between the previous deployment and the current deployment.

$$\% \text{ error} = \frac{\text{observed} - \text{baseline}}{\text{baseline}} \times 100 \tag{3}$$

Algorithm 1 General algorithm for locating a new deployment in a single QoS dimension

```
1:  procedure LOCATE(G, v, C, H, O, B)
2:      G: current graph of D&C solution space
3:      v: current location in graph G
4:      C: set of components in system
5:      H: set of hosts in system
6:      O: set of current observed performance for C
7:      B: set of baseline performance for C
8:      Δ = ∀c ∈ C : (o_c − b_c)/b_c × 100, o_c ∈ O, b_c ∈ B
9:      while Δ ≠ ∅ do
10:         Δ_i = max(Δ), i ∈ C
11:         if hist(G, i) < hist(G, (C − i)) then
12:             H' = H
13:             while H' ≠ ∅ do
14:                 h = candidate(H', v, Δ)
15:                 if H' − h ≠ H' then
16:                     e_{v,v'} = newedge(G, v, h, i)
17:                     if unique(G, v') then
18:                         accept(G, e_{v,v'})
19:                         return v'
20:                     end if
21:                     H' = H' − h
22:                 else
23:                     H' = ∅
24:                 end if
25:             end while
26:         end if
27:         Δ = Δ − Δ_i
28:     end while
29:     return NULL
30: end procedure
```

If the current deployment has improved QoS performance in a single dimension, such as end-to-end response time, we use Algorithm 1 to locate a new deployment and continuing improving QoS. As listed in Algorithm 1, given the current graph of the D&C solution space, hosts and components in the system, and baseline and observed performance, we find the component with the max percentage error (Line 8). We then determine if we can relocate the component based on its history (Line 11), such as how often has it been relocated to a new host or how has it performed in previous tests on a given host with other components. If we can relocate the component *i*, we find a candidate host *h* to relocate the component to (Line 14).

hist and *candidate* in Algorithm 1 are placeholders for a domain-specific function that understands how to evaluate the history of the tests and locate a candidate host based on the previous and current test, respectively. Moreover, the *hist* and *candidate* functions are used to prevent a greedy approach to frequently selecting the same component and host, respectively, as targets for relocation—which we call *deployment thrashing*—and exhaustively searching the solution space[1].

If the new deployment is unique in *G* (Line 17) and does not create a cycle, then we add it to the graph and return the new deployment (Line 19). Lastly, we execute the new deployment and repeat the location process for the new deployment. In the case that the current deployment does not improve the desired QoS, we backtrack to the previous deployment from the current deployment (vertex) and repeat the location process, *i.e*, Algorithm 1. We stop once we backtrack to the initial deployment and cannot locate a candidate component to reassign to a new host. This simple algorithm allows us—and in turn—developers to both understand the deployment solution space and locate solutions that meet QoS expectations.

4 Evaluating Our Deployment Search Technique Using the SLICE Case Study

In this section we present the results for searching the deployment solution space for the SLICE scenario using the search algorithm discussed in Section 3.

4.1 Experiment Design

The SLICE scenario introduced in Section 2.1 consists of 7 different component instances that must be deployed across 3 different hosts. Unlike any previous work that used the SLICE scenario [4], we do not place any constraints on how the components must be deployed onto their hosts. Instead, we are interested in locating deployments that yield better average end-to-end execution time for SLICE scenario's critical path—irrespective of the 350 msec deadline—based on Algorithm 1 discussed in Section 3.2.

The *hist* and *candidate* functions of Algorithm 1 are placeholders for domain-specific functions, *i.e*, functions that can better analyze the target domain and execution environment. Accordingly, we defined our history function for an individual component (Equation 4) and a set of components (Equation 5) as the percentage of times a component has been moved throughout the entire testing process.

[1] It is possible to design a history and candidate function that exhaustively searches the solution space, however, such functions are not ideal as the solution space grows larger and more complex.

$$hist(G,i) : \frac{\text{\# times } i \text{ moved}}{\text{\# of vertices in } G} \times 100 \quad (4)$$

$$hist(G,C) : \forall i \in C \frac{\text{\# times } i \text{ moved}}{\text{\# of vertices in } G} \times 100 \quad (5)$$

As shown later, this ensures that we do not constantly select the same component with the worst percentage error between unique deployments, and select components that may not have the worst percentage error in a test because it may not have been previously moved. The candidate function for selection is expressed in Equation 6,

$$candidate(H,v,\Delta) : min(avg(\Delta_{vh})) \quad (6)$$

where Δ_{vh} is the set of percentage error for components deployed on host $h \in H$ in deployment v, and identifies the host that contains the set of components with the least average percentage error from their baseline performance. This is conceptually similar to load balancing.

Table 2. Host specification for experiments.

ISISLab Host	Specification
1, 2, 3	Fedora Core 4, dual core 2.8 GHz Xeon 1 GB RAM, 40 GB HDD, 4 Gbps NIC

To test our search algorithm, we used the *Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS)* [4], which is a system execution modeling tool for component-based system. CUTS enables developers to rapidly construct high-level behavior models of a system using domain-specific modeling languages [6] and generate source code for a target component middleware that can be executed and monitored in the target environment. For the target execution environment, we used ISISLab `(www.isislab.vanderbilt.edu)` at Vanderbilt University, which is a cluster of computers managed and powered by Emulab [16] software. Table 2 lists the specifications of the three hosts we used in ISISLab for our experiments.

Table 3. Measured baseline of SLICE components.

Component	**Input**	**Output**	**Baseline (ms)**
(A) Config	Assess	Command	10
(B) EffMain (C) EffSec	Command	Status	10
(D) PlanOne	Track	Situation	45
(E) PlanTwo	Situation	Assess	45
(F) SenMain (G) SenSec	Command	Track	45

We used the three ISISLab hosts to measure the baseline performance of each component on each host in an isolated and controlled environment. This is necessary because each component's baseline performance is needed for Algorithm 1 in Section 3.2. Each baseline test was executed for 2 minutes, which resulted in 120 generated events handled by each component,

because 2 minutes was enough time for the component's baseline performance to stabilize.[2] Table 3 lists the baseline metric for each component by their input port (*e.g.*, event sink) and output port (*e.g.*, event source). We consolidated the baseline performances for each component on each host because they were equal.

4.2 Experiment Results

Interpreting the performance results. There are $3^7 = 2187$ unique deployments in the SLICE scenario. Table 4 lists the results for testing 17 unique deployments out of the 2187—for the SLICE scenario. The 17 unique deployments were derived using the search algorithm presented in Section 3.2 and Equations 4, 5, and 6. Out of the 17 unique deployments, we located 6 deployments (*i.e.*, tests 2, 4, 7, 15, 16, and 17) that improved the critical paths end-to-end execution time from the initial deployment.

Table 4. Results for searching SLICE deployment solution space.

Test	Deployment Strategy			Avg. End-To-End
	Host 1	Host 2	Host 3	Performance (ms)
1	B,D,F,G	A	C,E	182.7
2	A,B,D,F,G		C,E	181.2
3	A,B,F,G,D	E	C	185.9
4	B,C,D,F,G	C	E	180.2
5	A,D,F,G	C	B,E	186.7
6	A,B,D,G	C	E,F	186.5
7	A,B,D,F	C	E,G	180.4
8	A,B,F,G	C	D,E	193.3
9	A,B,C,D,F,G		E	182.5
10	B,D,F,G	C	A,E	189.6
11	A,D,F,G	B,C	E	206.9
12	A,B,D,G	F	C,E	207.9
13	A,D,F,G	B	C,E	189.8
14	A,B,F,G	D	C,E	188.8
15	A,D,B,F	G	C,E	178.9
16	A,B,D,F		C,E,G	179.1
17	A,B,D,E,F	G	C	174.6

The number of unique deployments in the SLICE scenario has never been exhaustively tested. We, however, can estimate how good a deployment is by summing the baseline performance for all the components in an execution path— assuming network and queuing performance is negligible since they represent another dimension of QoS—and comparing it against its observed performance. This summation represents the optimal performance for an execution path given there is no network contention and no queuing overhead.

The sum of the component's performance in the critical path for the SLICE scenario (see Section 2.1) is 155 msec. This is derived by summing the baseline performances for all the

[2] The duration of a baseline test depends on the complexity of the component's behavior, such as (non-)linear and conditional workflows

components in the critical path, which is given in Table 3. Test 4, 7, 15, 16, and 17 were the closest test to the optimal baseline end-to-end response time, *i.e.*, $< 16\%$ deviation, with test 17 being the closest, *i.e.*, a 12% deviation.

We also observed that for all 17 tests, the best end-to-end response time performance for the critical path was 169 msec (not present in Table 4). Even when all components were deployed onto the same hosts, such as in test 17, the end-to-end response time was greater than the optimal. This is, therefore, evidence of some form of software contention because the event generation rate for the SLICE scenario, *i.e*, 1 second, is not high enough to produce a workload that causes over-utilization of resources, such as the CPU.

Understanding host selection. Table 4 shows that a majority of the unique deployments did not utilize Host 2. This is attributed to Host 2 always having the worst average percentage error for each test. We did observe similar performance results in our baseline test, however, our technique tries to create deployments where components perform optimally (*i.e.*, close to their baseline) when collocated with other components. Moreover, our candidate function (Equation 6) was defined the avoid hosts that had components with a high percentage error from their baseline performance. We, therefore, did not investigate this anomaly and interpreted such results as recognizing and avoiding "bad" hosts that could potential worsen QoS performance.

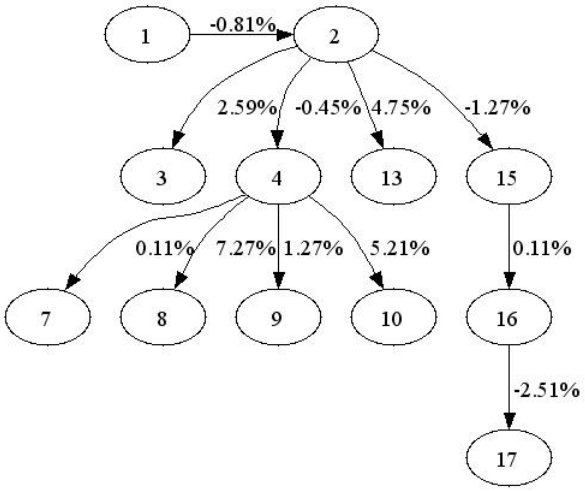

Fig. 6. Partial search graph for SLICE's unique deployments.

Interpreting the search graph. Figure 6 is the partial graph, or ordered tree, for the tests presented in Table 4. It illustrates the search process of Algorithm 1 using Equations 4, 5, and 6 for their corresponding placeholders in Algorithm 1. As illustrated in Figure 6, we first started with a random deployment of the SLICE scenario. The second test improves end-to-end response time for the critical path, however, the third test does not improve end-to-end response time. We, therefore, backtracked to the second test and selected a new component to relocate to a different host. This resulted in deriving test 4, which improved end-to-end response time for the critical path. Test 7 had an end-to-end response time close to test 4 (*i.e.*, 0.11% difference), but it was not accepted since it did not improve end-to-end response time.

It is, however, possible to use a standard error to determine the probability of accepting QoS performances that are close in value.

After test 7, we did not locate a new deployment that improved the critical path's end-to-end response time until test 15. This was a result of backtracking to the unique deployment for test 2, and selecting a new component to relocate to a new host. Consequently, we derived unique deployments for test 15, an improvement to test 2; test 16, an improvement to test 15; and test 17, an improvement to test 16.

We, therefore, can conclude that our algorithm can be parameterized with domain-specific functions to logically search the deployment solution space for a component-based system, and locate deployments that will improve QoS in a single dimension, such as average end-to-end response time, based on component baseline profiles. More importantly, it is possible to locate deployments that improve QoS in a single dimension based on a component's baseline profile.

5 Related Work

Traditional analytical techniques, such as rate monotonic analysis [14], queuing networks [15], and queuing petri nets [5], have been used to predict the performance of DRE systems. Such techniques enable system developers to predict performance at design time, however, they require in depth knowledge of the system architecture. Moreover, these techniques do not completely take into account the target operating environment, unless such a model exists and has been verified. Our technique differs from traditional analytical techniques because we evaluate a deployment by executing it in the target environment. We, therefore, are able to take into account the entire operating environment irrespective of system developers having any knowledge of its properties. We also believe that such analytical techniques can be integrated with our approach to analyze candidate deployments before executing them in the target environment to reduce search complexity and false positives.

Diaconescu et al. [2] discusses a framework and technique for improving the performance of component-based system by selecting implementations that are more suitable for a given execution context. Their technique involves "training" an adaptation module, locating performance anomalies in deployed applications, and select implementations that will address the anomaly and improve performance. Our technique is similar in that we use baseline profiles, which can be viewed as "training" to understand component performance. Our approach differs in that we do not focus on run-time adaption to improve performance. Instead, we are focus on locating a collection of unique deployments at design-time that will improve QoS performance, such as end-to-end response time.

Memon et al. [7] discuss a framework and customizable technique for providing continuous quality assurance for applications with a large configuration space. Their technique randomly tests different solutions in the configuration space and when a valid configuration, *i.e.*, one that meets their goal, is found, other configurations related to the good configuration is tested. Our approach is similar to their approach in that our search functions are customizable. Our approach differs in that we do not permutate all possible unique deployments and randomly test until we find a valid deployment. Instead, we start with a random deployment and base subsequent searches on the observations of the current test. Moreover, we base our search criteria on a component's percentage error from its baseline performance instead of testing by solely moving a component to a new host.

6 Concluding Remarks

Bad deployment choices for a component-based system can have dire effects on systemic QoS properties, such as end-to-end response time. In this paper, we presented a simple technique for locating deployments to improve system QoS along a single dimension of QoS. Our technique uses baseline profiles to calculate the percentage error in a component's performance irrespective of resource utilizations. This performance metric is used in conjunction with domain-specific functions to evaluate the history of a component and locate candidate hosts for relocating individual components based on a component's percentage error and current state of testing.

This approach allows for a flexible technique for searching the deployment solution space for systems that do not suffer from high resource utilization problems. Likewise, as more is learned about the system (and domain), system developers can modify their domain-specific functions to improve search capabilities.

Lessons Learned and Future Research

Techniques for searching deployment solution space is a process that continuously evolves as more is learned about the domain. The following list, therefore, discuss the lessons learned and future research directions:

- The SLICE scenario has 2187 unique deployments, however, the solution space has not been exhaustively tested. Future work, therefore, includes exhaustively testing the SLICE scenario's deployment solution space to locate the optimal deployments under various workloads, such as increased event generation rate, to strengthen the validation of our technique.
- Manually searching the deployment solution space, regardless of any search algorithm used, is inefficient and error prone. Future work, therefore, includes extending CUTS to automate the process of intelligently searching the solution space, which we believe will help address scalability challenges (*i.e.*, searching solution spaces where $|C|$ and $|H|$ are large numbers); however addressing scalability is highly dependent on the domain-specific functions ability to identify valid components and hosts for relocation.
- In some situations, exhaustively searching the deployment solution space many be less expensive than searching the solution space using Algorithm 1 in Section 3.2. This situation is highly dependent on the system under evaluation. Future work, therefore, including identifying and understanding situations where exhaustively searching the solution space is more favorable.
- Currently, we are testing unique deployments in an isolated environment. In a realistic environment, however, an application may be deployed with other applications. Future work, therefore, includes extending our technique to include situations of improving QoS in a single dimension when the DRE system comprises components belonging to multiple applications that are executing in the same environment.
- The generalized QoS problem has a N-dimensional solution space, however, we reduced the solution space to one dimension in this paper to simplify the deployment search problem and focus on the technique. Future work, therefore, will extend our technique to include multiple dimensions of QoS.

The CUTS framework and our research artifacts are available in open source at `www.dre.vanderbilt.edu/CUTS`.

References

1. Deng, G., Balasubramanian, J., Otte, W., Schmidt, D.C., Gokhale, A.: DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In: Proceedings of the 3rd Working Conference on Component Deployment (CD 2005), pp. 67–82. Grenoble, France (2005)
2. Diaconescu, A., Murphy, J.: Automating the Performance Management of Component-based Enterprise Systems Through the Use of Redundancy. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05), pp. 44–53 (2005)
3. Hill, J.H., Gokhale, A.: Model-driven Specification of Component-based Distributed Real-time and Embedded Systems for Verification of Systemic QoS Properties. In: Proceeding of the Workshop on Parallel, Distributed, and Real-Time Systems (WPDRTS '08). Miami, FL (2008)
4. Hill, J.H., Slaby, J., Baker, S., Schmidt, D.C.: Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In: Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications. Sydney, Australia (2006)
5. Kounev, S.: Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. IEEE Transactions on Software Engineering **32**(7), 486–502 (2006)
6. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer **34**(11), 44–51 (2001). DOI http://dx.doi.org/10.1109/2.963443
7. Memon, A., Porter, A., Yilmaz, C., Nagarajan, A., Schmidt, D.C., Natarajan, B.: Skoll: Distributed Continuous Quality Assurance. In: Proceedings of the 26th IEEE/ACM International Conference on Software Engineering, pp. 459–468. Edinburgh, Scotland (2004)
8. Menasce, D.A., Dowdy, L.W., Almeida, V.A.F.: Performance by Design: Computer Capacity Planning By Example. Prentice Hall PTR, Upper Saddle River, NJ, USA (2004)
9. de Niz, D., Rajkumar, R.: Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. International Journal of Embedded Systems **2**(3), 196–208 (2006)
10. Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. Ph.D. thesis, University College Dublin, Belfield, Dublin 4, Ireland (2007)
11. Rajkumar, R., Lee, C., Lehoczky, J., Siewiorek, D.: A Resource Allocation Model for QoS Management. In: In Proceedings of the IEEE Real-Time Systems Symposium (1997)
12. Shankaran, N., Koutsoukos, X., Schmidt, D.C., Gokhale, A.: Evaluating Adaptive Resource Management for Distributed Real-time Embedded Systems. In: Proceedings of the 4th Workshop on Adaptive and Reflective Middleware. Grenoble, France (2005)
13. Smith, C., Williams, L.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley Professional, Boston, MA, USA (2001)
14. Tri-Pacific: RapidRMA. www.tripac.com (2002)
15. Ufimtsev, A., Murphy, L.: Performance Modeling of a JavaEE Component Application using Layered Queuing Networks: Revised Approach and a Case Study. In: Proceedings of the Conference on Specification and Verification of Component-based Systems (SAVCBS '06), pp. 11–18 (2006)
16. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proc. of the Fifth Symposium on Operating Systems Design and Implementation, pp. 255–270. USENIX Association, Boston, MA (2002)