A Hierarchical Graph Model for Probing Multimedia Applications

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
bli@eecg.toronto.edu

Abstract

In order to achieve the best application-level Quality-of-Service (QoS), complex multimedia applications need to be dynamically tuned and reconfigured to *adapt* to unpredictable open environments offered by general-purpose systems. We believe that the objective of such adaptations should be to maintain a stable QoS with respect to a set of *critical* application QoS parameters. However, we have observed that only a limited set of parameters may be used as "tuning knobs" to affect the application behavior. In this paper, we present a *hierarchical graph model* to discover the relationships between the sets of *tunable* and *critical* QoS parameters. Based on such a model, we propose a polynomial-complexity QoS probing algorithm to quantitatively capture the run-time relationships between the two sets of parameters. Our probing algorithm is integrated into our broader framework, *Agilos*, which uses a configurable visual tracking application to verify the effectiveness of adaptations.

1 Introduction

In a best-effort open environment where general-purpose systems are used, complex multimedia applications may not be able to receive guaranteed Quality-of-Service (QoS). In such situations, they need to be dynamically tuned and reconfigured to *adapt* to the fluctuating environment, triggered by variations in resource availability. We claim that the objective of such adaptations is to maintain stability or optimality with regards to a set of *critical* QoS parameters within the application, since the QoS of these critical parameters will represent the overall quality and user satisfaction delivered by the application. On the other hand, any application has a set of *tunable* QoS parameters to serve as *tuning knobs* to tune and reconfigure the application's behavior. For example, in a distributed visual tracking application where multiple tracking algorithms are used to track the location of moving objects in a streamed live video, the *tracking precision* is the critical QoS parameter, where *frame rate*, *image compression ratio*, *number of simultaneous trackers* may all be tunable QoS parameters.

The introduction of critical QoS parameters as adaptation objectives leads to the problem of bridging the "gap" between two categories of parameters: critical and tunable QoS parameters. Critical parameters represent adaptation goals, and tunable parameters are the only "knobs" we may use. In order to reconfigure the set of tunable parameters with the goal of optimizing the quality of critical parameters, we need to discover the functional relationships between them. In this paper, we focus on this problem and present a hierarchical graph model to represent the dependencies between critical and tunable parameters, and present a *QoS probing algorithm* to quantitatively capture the run-time relationships between critical and tunable parameters. We have shown that with certain optimizations, the computational complexity of such an algorithm may be reduced from exponential to polynomial.

Previous work has focused on QoS-aware monitoring and probing mechanisms at the application level [1, 2, 3]. Al-Shaer *et al.* [2] has presented *HiFi*, an active monitoring architecture for monitoring distributed multimedia systems. Abdelzaher [1] has presented an on-line least squares estimator for estimating system parameters in QoS-aware web servers with a linear execution model. Chang *et al.* [3] has provided a *sandbox* implementation to tune resource availability when measuring application behavior. In comparison, our work do not assume prior knowledge of a particular execution model in an application, and focuses on discovering the relationships between parameters based on probing results, rather than the particular probing mechanisms. Finally, this work presents a polynomial-time algorithm to probe *multiple* critical parameters, which is an extension and major improvement over our own previous work [5].

The remainder of this paper is organized as follows. Section 2 presents formal definitions of different parameter categories. Section 3 describes our graph-based model to characterize relationships between different categories of application parameters, presents a QoS probing algorithm, and analyzes its computational complexity. Section 4 presents a brief case study with the visual tracking application, when integrating the probing algorithm into our broader control framework, *Agilos*. Section 5 concludes the paper.

2 Parameter Categorization

We consider a scheme of categorizing application QoS parameters. For this purpose, we focus on a single application component. We assume that this component accepts input with a QoS level $\mathbf{Q}^{in}$ and generates output with QoS level $\mathbf{Q}^{out}$, both of which are *vectors* of *application QoS parameter* values. In order to process input and generate output, a specific amount of resources $\mathbf{R}$ is required, which is a vector of resource types. Figure 1 illustrates such characterization in terms of QoS parameters and resources.

Formally, we define the vectors $\mathbf{Q}^{in}$, $\mathbf{Q}^{out}$ and $\mathbf{R}$ as follows:

$$\begin{aligned} \mathbf{R} &= [R_1, R_2, \ldots, R_m]^T \\ \mathbf{Q}^{in} &= [p_1^{in}, p_2^{in}, \ldots, p_n^{in}]^T \\ \mathbf{Q}^{out} &= [p_1^{out}, p_2^{out}, \ldots, p_k^{out}]^T \end{aligned} \quad (1)$$

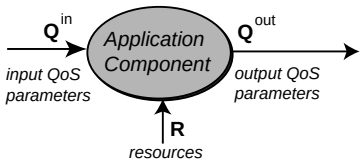

Figure 1: The Application and its Parameters

where $R_1, R_2, \ldots, R_m$ are the required resource types and measured with their respective units. In the tracking application, $m = 2$, and R_{cpu} is measured with CPU load percentage, while R_{net} is measured with bytes per second.

We assume the vector of all application QoS parameters $Q = [{Q^{in}}^T, {Q^{out}}^T]^T$. We further classify the parameters in the vector Q into three distinct categories:

- **Critical QoS parameters.** We assume all critical parameters, which are elements in the vector ${\mathbf{Q_{cr}}}^{out}$, belong to the set of output QoS parameters. Let ${\mathbf{Q_{cr}}}^{out} = [p_{i_1}^{out}, p_{i_2}^{out}, \ldots, p_{i_l}^{out}]^T$, while $1 \leq i_1, i_2, \ldots, i_l \leq k$. The objective of adaptation is focused on these critical QoS parameters.

- **Tunable QoS parameters.** Without loss of generality[1], we assume that all input QoS parameters are *tunable*.

- **Non-critical QoS parameters.** It follows that any parameters in the vector $\mathbf{Q}^{out}$ that do not belong to the category of *critical QoS parameters* are non-critical.

For simplicity of notations, we redefine $\mathbf{Q}^{in}$ and ${\mathbf{Q_{cr}}}^{out}$ as follows, removing extra superscripts and renumbering the subscripts:

$$\begin{aligned} \mathbf{Q}^{in} &= [p_1, p_2, \ldots, p_n]^T \\ {\mathbf{Q_{cr}}}^{out} &= [p_1^{cr}, p_2^{cr}, \ldots, p_l^{cr}]^T \end{aligned} \tag{2}$$

Since all tunable QoS parameters are input parameters and all critical QoS parameters belong to the set of output QoS parameters, it is natural that when the tunable parameters are actively controlled, the critical parameters will consequently change. In this case, the critical parameters are claimed to be *dependent* on the tunable parameters. In addition, in most applications critical parameters are also *dependent* on a certain subset of resource types, since when resource availability changes, they effectively change the observed values of critical parameters.

In order to characterize the complete set of parameters that critical parameters depend on, we define a new vector $\mathbf{P}^{in}$ to include the relevant resource types:

$$\mathbf{P}^{in} = [{\mathbf{Q}^{in}}^T, \mathbf{R}^T]^T = [p_1, p_2, \ldots, p_n, R_1, R_2, \ldots, R_m]^T \tag{3}$$

For coherent notations, we redefine $p_{n+i} = R_i$, $1 \leq i \leq m$, so that:

$$\mathbf{P}^{in} = [p_1, p_2, \ldots, p_n, p_{n+1}, \ldots, p_{n+m}]^T \tag{4}$$

[1] If an input QoS parameter is not tunable, we may view it as an output parameter instead.

3 Probing Application QoS Parameters

In this section, we present a QoS probing algorithm that bridges the gap between the set of critical and dependent parameters by capturing the dependency relationship between ${\mathbf{Q_{cr}}}^{out}$ and $\mathbf{P}^{in}$.

3.1 A Bipartite Graph Model

The brute-force way of designing a QoS probing algorithm is to discover the relationship between any particular critical parameter and all its dependent parameters. For this purpose, we need to construct a *Directed Bipartite Graph*, with all elements in ${\mathbf{Q_{cr}}}^{out}$ forming one partition of the graph, and all elements in $\mathbf{P}^{in}$ forming the opposite partition. If node p_i^{cr} depends on p_j, $1 \leq i \leq l, 1 \leq j \leq n+m$, there exists a directed edge from node p_i^{cr} to node p_j in the directed bipartite graph.

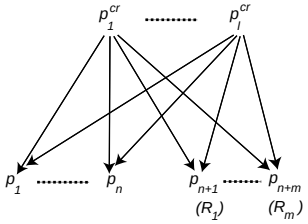

Figure 2: The Bipartite Graph Model for Application QoS Parameters

The objective of a QoS probing algorithm is to tune the parameters in $\mathbf{P}^{in}$ and observe the values of critical parameters. As an initial step, we assume that each critical parameter in ${\mathbf{Q_{cr}}}^{out}$ depends on all the parameters in $\mathbf{P}^{in}$. Figure 2 shows an example. Obviously, the subgraph consisting of one critical parameter and all parameters in $\mathbf{P}^{in}$ is a *two-level tree*, with the selected critical parameter as root and all the parameters in $\mathbf{P}^{in}$ as leaves.

The critical step in the QoS probing algorithm is to discover the relationship between dependent nodes. For this purpose, we assume that for $\forall i, \forall t$, there exists $\{p_i\}_{min}$ and $\{p_i\}_{max}$ such that $\{p_i\}_{min} \leq p_i(t) \leq \{p_i\}_{max}$, any value beyond this range is either not possible or not meaningful. For example, the *frame rate* may vary in the range of $[1, 30]$ (in frames per second). Hence, the dependency between each critical parameter and their dependent parameters can be characterized by f_i, defined as:

$$\begin{aligned} p_i^{cr} &= f_i(p_1, p_2, \ldots, p_{n+m}) \\ \{p_k\}_{min} &\leq p_k(t) \leq \{p_k\}_{max} \\ k &= 1, 2, \ldots, n+m, i = 1, 2, \ldots, l \end{aligned} \tag{5}$$

With the Bipartite Graph Model, the probing algorithm for computing the dependency relationship between each critical parameter p_i^{cr} and the parameters in $\mathbf{P}^{in}$ is straightforward. For each critical parameter p_i^{cr}, It consists of $n+m$ **for** loops, each one of them iterating through the range of $n+m$ leaf parameters.

Theorem 1. the computational complexity of the probing algorithm based on a Bipartite Graph Model is $O(l * {N_{max}}^{(n+m)})$, where $N_{max} = \max\{N_{p_1}, N_{p_2}, \ldots, N_{p_{n+m}}\}$, and $N_{p_j} = (\{p_j\}_{max} - \{p_j\}_{min})/\{p_j\}_{increment}$.

Proof. Omitted for space limitations.

This shows that when the number of tunable parameters and resource types increases, computation increases *exponentially*. In order to carry out QoS probing on-line, we need a more efficient algorithm.

3.2 A Hierarchical Graph Model

Typically, one critical parameter only depends on a limited number of tunable QoS parameters and resources. Furthermore, we have observed that if two critical parameters depend on a common set of parameters in $\mathbf{P}^{in}$, they may have similar dependency relationships with such a common set. If this similarity of relationships can be captured and then shared by both critical parameters, computational complexity may be reduced dramatically.

We introduce a set of *intermediate QoS parameters* that critical parameters depend on. These intermediate QoS parameters may be either non-critical QoS parameters in the output QoS, or other internal parameters within the application component. In addition, they depend on the parameters in $\mathbf{P}^{in}$. To maximize the dependency relationship sharing among critical parameters, intermediate nodes are organized hierarchically. Figure 3 shows an instance of the possible dependency relationships.

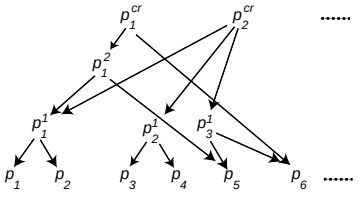

Figure 3: Hierarchical Graph Model for Application QoS Parameters

The hierarchy in Figure 3 presents the following properties:

- The subgraph composed of a critical parameter node p_i^{cr} $(1 \leq i \leq l)$ and all downstream nodes is essentially a multi-level directed tree, with root as p_i^{cr} and a subset of parameters in $\mathbf{P}^{in}$ as leaves. We refer to such a tree as Tree(p_i^{cr}), and the hierarchical graph is $\cup\{\text{Tree}(p_i^{cr})\}$.
- Two critical parameters share dependency on some parameters by sharing a subtree. For example, in Figure 3, the subtree with source at p_1^1 is shared by both critical parameters p_1^{cr} and p_2^{cr}.
- All nodes other than leaves have at least outdegree 2. Otherwise, redundant nodes may be removed as shown in Figure 4.

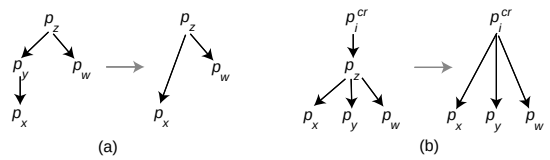

Figure 4: Redundant node removal

- The notation p_i^j denotes that the node is at level j, and is the i^{th} node at this level. We calculate the *level* of a node by counting from bottom, with leaves (parameters in $\mathbf{P}^{in}$) at level 0, that is, $p_i = p_i^0, i = 1, 2, \ldots, n+m$.

Assume the parent node p_i^j depends on k child nodes $p_{i_1}^{j_1}$, $p_{i_2}^{j_2}$, ..., $p_{i_k}^{j_k}$. The dependency can thus be characterized by a function $f_{p_i^j, p_{i_1}^{j_1}, p_{i_2}^{j_2}, \ldots, p_{i_k}^{j_k}}$, defined as:

$$p_i^j = f_{p_i^j, p_{i_1}^{j_1}, p_{i_2}^{j_2}, \ldots, p_{i_k}^{j_k}}(p_{i_1}^{j_1}, p_{i_2}^{j_2}, \ldots, p_{i_k}^{j_k}) \quad (6)$$
$$\{p_{i_s}^{j_s}\}_{min} \leq p_{i_s}^{j_s}(t) \leq \{p_{i_s}^{j_s}\}_{max}$$
$$s = 1, 2, \ldots, k$$

A hierarchical QoS probing algorithm is shown in Figure 5. For each critical parameter, we calculate all dependency functions between one non-leaf node and its dependent child nodes. The calculation is performed from bottom to top of the hierarchical graph. The idea in this algorithm is that, if a child node p_k^j is not a leaf, it must depend on some other lower level nodes, and the dependency between them should have already been calculated and saved in the log. The calculated value set for this node should be within the range $[\{p_k^j\}_{min}, \{p_k^j\}_{max}]$, and each value is rounded to the nearest discrete value $\{p_k^j\}_{min} + s * \{p_k^j\}_{increment}, 0 \leq s \leq N_{p_k^j}$, where $N_{p_k^j} = (\{p_k^j\}_{max} - \{p_k^j\}_{min})/\{p_k^j\}_{increment}$. The sample values for this child node will be retrieved from the log.

for each critical parameter p_i^{cr} and its associated Tree(p_i^{cr})
  for level j = 1, $j \leq$ depth(Tree(p_i^{cr})), j ++
    for each node p_k^j, whose children are $p_{k_1}^{j_1}, p_{k_2}^{j_2}, \ldots, p_{k_d}^{j_d}$
      d := outdegree(p_k^j)
      list := $\{p_k^j\}$ // observing list
      for each child $p_{k_w}^{j_w}$, $1 \leq w \leq d$
        if $p_{k_w}^{j_w}$ is a leaf and is *not* a resource **then**
          $p_{k_w}^{j_w}$ iterates from $\{p_{k_w}^{j_w}\}_{min}$ to $\{p_{k_w}^{j_w}\}_{max}$, step $\{p_{k_w}^{j_w}\}_{increment}$
        else if $p_{k_w}^{j_w}$ is a resource parameter **then**
          if $p_k^j \in$ *list* **then**
            list := *list* - $\{p_k^j\}$
            list := *list* + $\{p_{k_w}^{j_w}\}$
            p_k^j iterates from $\{p_k^j\}_{min}$ to $\{p_k^j\}_{max}$, step $\{p_k^j\}_{increment}$
          else
            list := *list* + $\{p_{k_w}^{j_w}\}$
        else // non-leaf node
          search *log* and find the sorted value set for $p_{k_w}^{j_w}$
          $p_{k_w}^{j_w}$ iterates elements in the found value set
        log observed values for parameters in *list*

Figure 5: The Hierarchical QoS Probing Algorithm

Note that if a resource parameter is one of the dependent nodes, we control the values of the parent node instead and observe the changes of the resource parameter. The variable *list* keeps track of the parameters to be observed. This is because resource usage is usually hard to control precisely.

According to the algorithm, it is obvious that if a node in Tree(p_i^{cr}) and a node in Tree(p_j^{cr}) share a subtree, the calculation for the subtree may be done only once for both trees and the dependency function can thus be shared by both nodes. For example, in Figure 3, the subtree rooted at p_1^1 exists in both Tree(p_1^{cr}) and Tree(p_2^{cr}). After the dependency function $f_{p_1^1, p_1, p_2}$ is determined, it will be shared by parent nodes p_1^2 in Tree(p_1^{cr}) and p_2^{cr} in Tree(p_2^{cr}).

3.3 Complexity Analysis

The computation in the algorithm is sequentially performed for each Tree(p_i^{cr}), $i = 1, 2, \ldots, l$. Therefore, we first consider a single tree Tree(p_i^{cr}). It is obvious that the complexity depends on the number of non-leaf nodes in the tree. We first demonstrate that the number of non-leaf nodes is bounded by the number of leaves in Tree(p_i^{cr}).

Let $W(T)$ and $L(T)$ denote the number of non-leaf nodes and number of leaves in a Tree T, respectively. Let $h(T)$ denote the depth of the tree T.

Lemma 1: Given a directed tree T. If outdegree for each non-leaf node is no less than 2, then $W(T) < L(T)$.

Proof. Proved by induction. Omitted for space limitations.

For each node p_k^j in Tree(p_i^{cr}), we assume that its outdegree is $d(p_k^j)$. Let $d_{max}^i = \max\{d(p_k^j)\}$, the maximum outdegree of the nodes in Tree(p_i^{cr}). Let $d_{max} = \max\{d_{max}^i\}$, $1 \leq i \leq l$, the maximum outdegree of the nodes in $\cup\{\text{Tree}(p_i^{cr})\}$. In addition, let $N_{max} = \max\{N_{p_k^j}\}$.

Theorem 2: The computational complexity of the hierarchical QoS probing algorithm is bounded by $O(l * (n + m) * (N_{max})^{d_{max}})$. l is the number of critical parameters, n is the number of tunable parameters and m is the number of resource parameters.

Proof. For Tree(p_i^{cr}) in the hierarchical graph $\cup\{\text{Tree}(p_i^{cr})\}$, we count the number of non-leaf nodes in this tree. According to *Lemma 1*, the number of non-leaf nodes is less than the number of leaves, which is $n + m$. Therefore, the total time of profiling one critical parameter is less than $(n + m) * (N_{max})^{d_{max}}$. For l critical parameters, the upper bound is $O(l * (n + m) * (N_{max})^{d_{max}})$. This is obviously *polynomial*. **QED.**

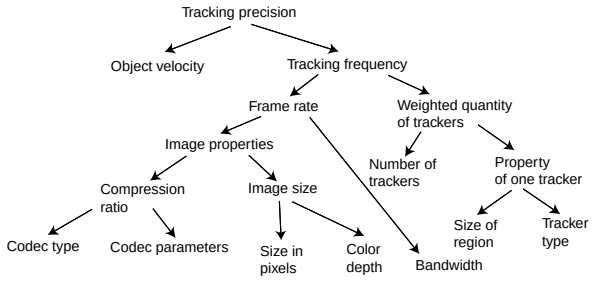

Figure 6: Hierarchical Graph for Distributed Tracking

4 Case Study: Distributed Visual Tracking

In our experiments, the QoS probing algorithm presented in this paper has been integrated into our broader QoS control framework, the *Agilos* architecture [4]. Figure 6 shows the hierarchical graph for the tracking application. Discovered by the probing algorithm, Figure 7 shows the dependency relationships between the critical parameter, *tracking precision*, and the intermediate QoS parameters *object velocity* and *tracking frequency*. We have shown in [4] that the *Agilos* architecture is able to control the adaptation behavior in the tracking application, so that the *tracking precision* remains stable at all times, even with the presence of bandwidth and CPU fluctuations due to a best-effort execution environment. The QoS probing algorithm plays an important role in this framework.

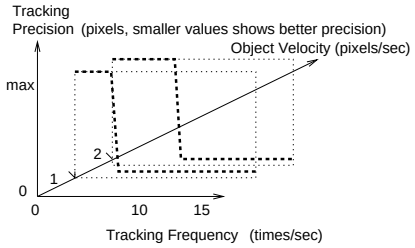

Figure 7: QoS Probing Results: An Example

5 Conclusion

In this paper, we first define application QoS as the quality of a set of *critical* parameters. Based on such assumptions, we have presented a polynomial-time QoS probing algorithm for probing multimedia applications, and discovering functional relationships between critical and tunable parameters. This algorithm has been integrated into the broader *Agilos* framework for controlling applications to adapt to best-effort open environments. A visual tracking application is deployed under the control of *Agilos*, and the *critical* parameter, tracking precision, remains stable at all times by trading off other non-critical parameters.

References

[1] T. Abdelzaher. An Automated Profiling Subsystem for QoS-Aware Services. In *Proceedings of Second IEEE Real-Time Technology and Applications Symposium*, 2000.

[2] E. Al-Shaer, H. Abdel-Wahab, and K. Maly. HiFi: A New Monitoring Architecture for Distributed Systems Management. In *Proceedings of IEEE ICDCS 99*, pages 171–178, May 1999.

[3] F. Chang and V. Karamcheti. Automatic Configuration and Run-time Adaptation of Distributed Applications. In *Proceedings of HPDC-9*, pages 11–20, August 2000.

[4] B. Li, W. Jeon, W. Kalter, K. Nahrstedt, and J. Seo. Adaptive Middleware Architecture for a Distributed Omni-Directional Visual Tracking System. In *Proceedings of the SPIE Multimedia Computing and Networking 2000*, pages 101–112, January 2000.

[5] B. Li and K. Nahrstedt. QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications. In *Proceedings of Middleware 2000*, pages 256–272, April 2000.