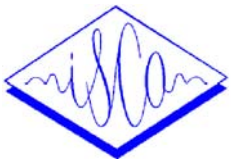

INTERSPEECH 2004 - ICSLP
8th International Conference on Spoken Language Processing
ICC Jeju, Jeju Island, Korea
October 4-8, 2004

Automatic Network Optimization of Voice Applications

Chaitanya Ekanadham [1], *Juan M. Huerta* [2]

[1] Stanford University
Stanford CA 94309
chaitu@stanford.edu

[2] IBM T J Watson Research Center
Yorktown Heights NY 10598
huerta@us.ibm.com

Abstract

We present a technique to automatically transform a voice application's call flow, or interaction graph, into the set of application pages that encapsulate the call flow nodes and resources needed to run such application in a client-server environment. Our technique, called NOVA, performs this automatic partition by making use of a set of cost functions that model the latencies associated with the transmission of data and application resources through a network and their processing by the client components (*e.g.*, voice browsers, grammar compilers and system engines). Our technique facilitates the existence of tools that permit application developers to focus on designing the call-flow of the application while leaving the task of segmenting and packaging the application into pages to our algorithm. The cost functions utilized by our algorithm can be dynamically computed allowing for runtime application optimization. We demonstrate the impact of our algorithm through simulation experiments.

1. Introduction

Speech applications today are widely used in client-server Web environments similar to that shown in Figure 1. In such environments, an application server presents the application markup and resources (the grammars, prompts, and markup code) to the client (voice browser) through a network after the client requests these resources. The end-user interacts with the application through a user interface, which in the case of voice applications typically consists of a telephony link. The goal of the application is, typically, to execute a transaction (*e.g.*, access a database and provide information to the user) after sufficient information has been obtained from the user.

Designing, building and deploying speech applications today require a team of specialists that focuses on separate sections of the development process. Typically, a speech application is designed by specifying the interactions required with the user, then implementing such specification into the corresponding markup or into code that renders such markup (*e.g.,* [1]). The goal of the backend is to execute the transaction based on the input data provided by the user. Finally, once the backend and the application code are in place, the system is tuned for performance to make sure that the error rates, rejection thresholds, latency times, interface and other performance metrics of the system are acceptable. If changes need to be made to the application, the call-flow, code implementation, or application parameters are adjusted and the application is tested again until the acceptable behavior is shown.

The skill-set required to design, build and tune such applications is heterogeneous: HCI experts design the interaction call-flow and prompts, application developers write the code that implements the call-flow and package the application, and speech scientists tune the grammars, pronunciations, rejection thresholds, and any other parameter affecting recognition accuracy and performance.

Several tools exist today to support the multiple phases of voice application development (*e.g.*, TellMe Studio, VoiceToolkit, WSAD [2, 3, 4]). Many of these tools support automatic or semiautomatic code generation (for example, the automatic generation of client VXML markup) from the call-flow aiming to simplify the application development cycle.

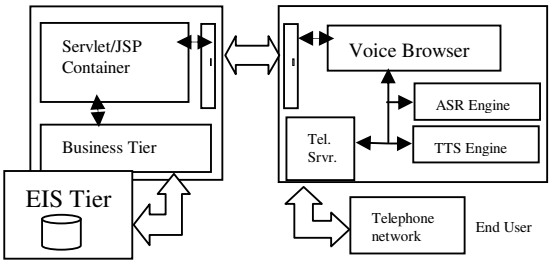

Figure 1. Schematic representation of the typical client-server voice application deployment environment.

In this paper we address the problem of simplifying the speech application development cycle for the client-server framework shown above by developing intelligent mechanisms that provide added value to tool offerings. Specifically, we present a method called NOVA that automatically partitions the application flowchart (or call-flow) into the pertinent *pages*. A page is the set of resources that are sent to the client in a single request. As we discuss below, the partition found by NOVA is based on a cost function that reflects the cost of transmitting and processing data in the configuration above. Our technique can be applied on the static call-flow prior to deploying the application, or during run-time to dynamically adjust the application to the varying conditions of the browser and network. When applied on the static call-flow, NOVA can be employed in conjunction of automatic client markup generation algorithms to generate the deployment archive file from the call-flow.

This paper is organized as follows. In Section 2, we describe in more detail the voice application development process and the current tools that support it. In Section 3, we further explain NOVA and describe methods in which it can be incorporated into tooling. In Section 4, we present the results of our experiments evaluating NOVA's benefits.

2. Voice Application Development and supporting tooling

As outlined in Section 1, the development process of a voice application consists of the following steps:

Call-flow specification: The application's user interaction is designed and specified in a graph called the call-flow. Prompts and interaction mechanisms are generally specified here.

Application Coding: the call-flow is transformed into client markup or into markup-generating code (*e.g.*, JSP tags [5]). The application data model and backend are also typically developed in this phase.

Application packaging and deployment: the application's markup or its tags, is embedded in pages and the controller artifacts and other supporting deployment files are put into place. The application deployment archive file is generated.

Application testing and tuning: the application is deployed, tested and its parameters tuned.

To establish the potential impact of our proposed technique on tooling, we describe below the existing tooling that facilitates these development steps for each of the processes described above.

Tooling for Flowchart design: Many tools allow for the design of the call flow by a graphical (GUI) interface. In this tooling modality, the interaction nodes are selected from a palette and connected into a graph [3]. The interaction nodes or controllers for speech correspond loosely to input fields, decision nodes, processing nodes and prompt nodes. The interaction nodes also need to have their attributes specified. Some of these attributes refer to resources: grammars, prompts, pronunciation lists. Some existing tools also have the capability to generate the client markup automatically.

Tools for Web Application coding: These tools aid in the implementation of various application supporting files: data model, controller code, as well as the manual development of code from the flowchart. Depending on the language chosen and the framework, framework-specific tooling will have language editors which aid in the development and packaging of these files (*e.g.,* WSAD [4]).

Application packaging, deployment and tuning: The modularization step involves writing the controller code and artifacts including configuration and descriptor files. Some tools automatically generate control code from a control specification. Tuning is typically done by evaluating the application in controlled deployments.

The method we propose in this paper aims at linking the call-flow design and automatic markup code generation steps with the application segmenting and packaging in a way that optimizes the application's expected average latency. By allowing this, the call flow designer (typically a Human Computer Interaction specialist, and not a web developer or programmer) would single-handedly be able to design, deploy, and test actual applications with ease.

3. Automatic Network Optimization of Voice Applications

3.1. Overview

Our algorithm, known as *NOVA* (Network Optimization of Voice Application), takes as input the directed graph, or node-based representation of a voice application, as well as parameters that affect network latency and produces a *partition* that minimizes the average application latency. A partition of the application defines which parts of the application should be packaged together as a *page*, and must therefore be sent to the client when a single request is made. NOVA takes into account all possible traversals through the application and their respective likelihoods of occurrence (this information is given through probability weights on the arcs of the call flow graph) and minimizes the *average latency* for the entire application.

Two constraints imposed on a group of nodes representing an application page are that they must: 1) be rooted and connected directed graphs themselves, and 2) nodes designated as *break* nodes (nodes at which the developer indicates a mandatory roundtrip to the server) are always terminal nodes in their page.

3.2. Transmission of data between client and server

To illustrate the role of NOVA, consider the simplified application call-flow show in the left panel in Figure 2. The dotted line represents the path that a user could follow in a sample call. If the whole graph was encoded into markup and written in a single page then the markup and resources for the whole application would be transmitted by the server to the client when the call is initiated. The total time for the call, which we call the application latency, is the total time it takes for 1) the request to go to the server, 2) the server to produce the page, 3) the resources to be transmitted, 4) the resources to be compiled and processed by the different engines, 5) the user to interact with the application, and 6) the resulting data to be submitted back to the server. For this simple example we can see that several unused nodes were sent to the server affecting the transmission and processing time. NOVA aims to reduce average application latency by segmenting the application into reasonable partitions.

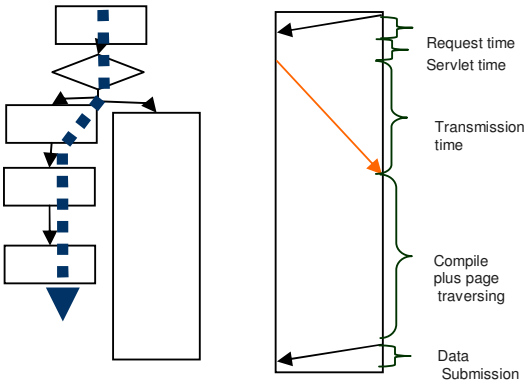

Figure 2. Application call-flow denoting a graph traversal path, and client-server roundtrip

3.3. The Application Graph

There is information about the application graph which must be supplied to NOVA for it to obtain the optimal partition. This is: 1) the *cost* of each node in the graph related to its relative resource's sizes (*e.g.,* grammars,

prompts, markup), 2) the transition probabilities between nodes and 3) nodes that are break nodes should be explicitly labeled so.

3.4. Cost Function

We now focus on three functions which affect the application's latency. They are the latency incurred from: 1) processing and compiling an application page by the client (*e.g.*, the compilation of grammars by the voice browser, processing of prompts by TTS engine, compilation of markup code), 2) the transmission of an application page from the server to the client, and 3) transmitting any needed information from the client back to the server. Let us call these functions, *f, g,* and *k·g*, respectively (assuming that the amount of data returned to the server is some fraction *k* of the amount of data contained in the page that collected this data). Also, let C_x denote the page containing a node *x* (*e.g.*, $f(C_x)$ represents the time it takes to process and compile the page containing node *x*). Note that the *traversal time*, or the time taken by the user to interact with the application, is constant for all possible partitions of the application graph, and can therefore be ignored.

Thus, latency is incurred only at each *transition* in the application, or point at which data is exchanged between client and server, as well as at the beginning of the application (processing of the first page). The arcs leaving the initial, or *root* node (which can simply correspond to a user's initial request) are *rooted arcs*. A *critical arc* is an arc where a roundtrip to the server is made (*e.g.*, arcs connecting nodes of different pages). Let an arc connecting two nodes *a* and *b* be denoted *(a,b)*, and let the sum of the probabilities of all paths containing *(a,b)* be denoted by *P(a,b)*. Equation 1 shows the proposed expression for the average latency of an application, for a given graph partition with *m* critical arcs and *n* rooted arcs.

Equation 1: Cost Function Expression for $L_{Application}$

$$L_{Application} = \sum_{i=1}^{m} P(a_i,b_i) \cdot \left[k \cdot g\left(C_{a_i}\right) + g\left(C_{b_i}\right) + f\left(C_{b_i}\right)\right] + \sum_{j=1}^{n} P(a_j,b_j) \cdot f\left(C_{a_j}\right)$$

3.5. Computation Storage Stacks

NOVA uses a node property known as *color* to denote to which page a certain node belongs. Thus, all nodes of the same color represent a single page. For a graph with *N* nodes, the maximum possible number of distinct colors is *N*. NOVA computes the optimal partition by associating each node to a color. To achieve this, NOVA uses four computation storage stacks of size *N*, as shown in Figure 3.

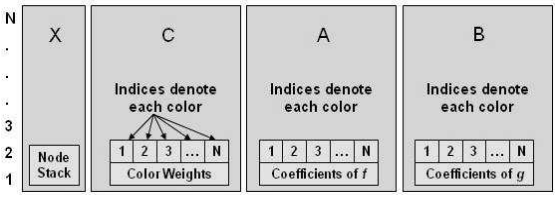

Figure 3. Computation storage stacks.

Stack *X* in Figure 1 keeps track of which nodes have been *colored* so far. Nodes are added to stack *X* in topological order and assigned a new color. The stack is then scanned top to bottom for any new critical or rooted arcs between stacked nodes that may be formed from the new addition. If these exist, they are used to *increment* the cumulative values at the indices (denoting each color) of the topmost element in stacks *C*, *A*, and *B*. Thus, at any point when the stack X contains *R* nodes (index *R* is the size of all the stacks, so let the topmost element of any stack *Z* be denoted by Z_R), the average latency of stack *X* (its measure of latency, L_X) can be computed using Equation 2:

Equation 2*:* Adapted Cost Function

$$L_X = \sum_{k=1}^{N} \left\{ \left(A_R[k] \cdot f\left(C_R[k]\right)\right) + \left(B_R[k] \cdot g\left(C_R[k]\right)\right) \right\}$$

Once the stack is full, the expression in Equation 2 is equivalent to the average latency for the finished partition. A new minimum latency is *guaranteed* each time the stack is full because, if at any point the computed score of the stacks exceeds the best score so far, the top node is *recolored*, or assigned a different color. The color to be assigned is picked by scanning the stack from top to bottom for parents of the top node (that are not designated *break nodes*) and assigning the top node the color of the first parent found in the stack that has not already been matched in this manner. If no parents that have not already been matched can be found in the stack, the top node is removed from the stack and its color property is removed, and the new top node is then recolored. This continues until the stacks are empty. The pseudo-code in Figure 4 summarizes the coloring procedure.

```
while (stack X is not empty) {
    if (stack X is full) {
        bestPartition = stack X
        bestScore = computeScore( )
    } else {
        addToStackX( nextNodeInTopologicalOrder( )
)
        updateStacks(C,A,B)
        if (computeCost( ) < lowestCost)  { continue }
    }
    while (stack X is not empty) {
        Recolor(top node in stack X)
        updateStacks(C,A,B)
        if ( (Recolor fails) | | (computeCost( ) >
        lowestCost) ) {
            removeFromStackX( top node )
            updateStacks(C,A,B)
        } else { break }
    }
}
```

Figure 4. Pseudo-code for NOVA algorithm

4. Experiments

To assess the value of our algorithm we performed several simulation experiments. In these simulations, we have assumed that the transition probabilities and the weights of the nodes correspond to Independent Random

Variables with mean *m* and variance *v* to generate various graph instances. Our goals are to: (a) show the importance of NOVA by observing the distribution of resulting average latencies for all the partition of a given graph, and (b) see how efficient NOVA is in computing the best partition. For simplicity, in all our simulations we assumed a quadratic function that relates the processing and compilation time of the resources. This function can be chosen to be more complex to reflect more complex environments, yet the efficiency of the algorithm is not dependent on this function. We assumed a similar quadratic function for the transmission delay.

```
F(x) =0.007 x^2 + 0.92 x + 0.17
```

Under normal conditions, an application graph only shows the possible nodes and transitions it implements. However, in real applications, a given user traverses the graph in a particular way, and the collection of all traversal routes happening can be described in terms of probability distributions. If a partition is selected at random, how much will the performance suffer on average as compared to the best partition?

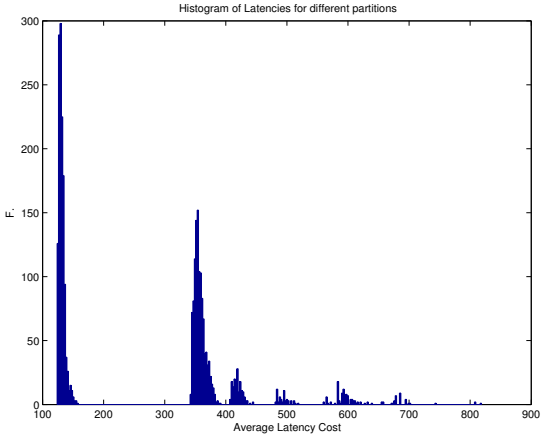

Figure 5. Histogram of average latencies for all partitions of a given graph

To answer the question above and thus show the importance of NOVA, we computed the distribution of resulting average latencies for all the partition of a given graph, and a set of parameters. Figure 5 above shows the histogram of such latencies for a graph containing 8 nodes. We can see that the values of latencies go from above 120 to around 800. This means that if we were to choose a random partition, we could conceivably end up observing a resulting average latency whose value is 200% or 300% the optimal value, which shows the need for an algorithm like NOVA.

To asses NOVA's computational efficiency, we generated random application graphs of increasing size g_n. For each of these graphs we computed the total number of possible partitions *Partitions*(g_n) as well as the running time of a constrained search by NOVA (*NovaTime*(g_n)). In this constrained search modality, the algorithm abandoned any partial coloring that was worse than the best scoring partial coloring at any given time. Figure 6 shows the ratio between *NovaTime*(g_n) and *Partitions*(g_n), reflecting the fact that while the running time of NOVA and the number of colorings grew as a function of the number of nodes, the ratio of the time over the number of partitions decreased. This means that the number of partitions increases at a faster rate than NOVA's running time per partition. This reflects the efficiency of the algorithm in searching for the best partition in the growing space of possible colorings.

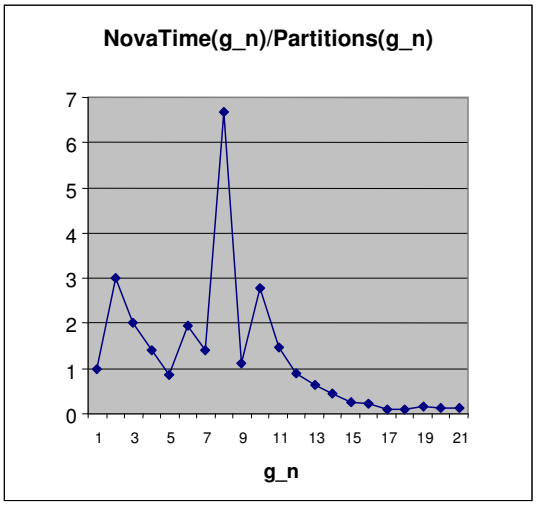

Figure 6. Ratio of average latency for NOVA partition vs. number of partitions for graphs of various sizes.

5. Conclusions

We have described the voice application development process and proposed a technique that can bridge the gap between call-flow design and code generation and packaging of the application into an optimal partition using a set of cost functions. We showed that even for simple graphs, the impact of a poor partition choice can severely impact the average latency of the application. Our technique finds a graph's best average latency in an efficient way. Because of this efficient ability to fragment an application graph into the best partition, NOVA can thus constitute the building block for intelligent tools that allow rapid application design, prototyping and deployment. Furthermore, there are conceivable situations in which NOVA is deployed in a network environment. Based on the dynamic real-time changing conditions of the network and client/server, NOVA could modify the application segmentation to allow for better system response allowing for autonomic application tuning.

6. References

[1] W3C 2004 Voice Extensible Markup Language
http://www.w3.org/TR/2004/REC-voicexml20-20040316
[2] TellMe Studio
http://studio.tellme.com/
[3] IBM. Voice Toolkit
http://www.alphaworks.ibm.com/tech/voicetoolkit
[4] IBM. WebSphere Studio Application Developer
http://www-306.ibm.com/software/awdtools/studioappdev
[5] Sun Micro Systems. Java Server Pages 2.0
http://java.sun.com/products/jsp