Efficient Retargeting of Generated Device User-Interfaces

Olufisayo Omojokun
Georgia Institute of Technology
omojokun@cc.gatech.edu

Prasun Dewan
University of North Carolina at Chapel Hill
dewan@cs.unc.edu

Abstract

Many pervasive computing systems have been built for using mobile computers to interact with networked devices. To deploy a device's user-interface, several systems dynamically generate the user-interface on a mobile computer. While this approach has several advantages, empirical results from different generators show that it takes a relatively long time for a mobile computer to create a user-interface from scratch. This paper shows that it is possible to overcome this limitation by efficiently mapping or retargeting a previously generated user-interface of one (source) device to another (target) device of the same or different type. Using the implementation of an existing generator and a set of real-world scenarios, we show that user-interface retargeting can yield deployment times that are often as good as or noticeably better than the approach of locally loading pre-existing manually-written user-interface code.

1. Introduction

One of the visions of ubiquitous computing is using mobile computers to interact with networked devices. Realizing this vision would allow, for example, a person in a wheel chair equipped with a mobile computer to interact with hard to reach elevator and door controllers. The person could similarly use the same mobile computer to control projectors, lights, and A/V equipment in a presentation room.

An important issue raised by this vision is: How should the user-interfaces of target devices be deployed on mobile computers? One approach involves executing manually-written preinstalled user-interface code on a mobile computer's local storage. To illustrate, the wheel chair's owner would pre-install user-interface programs for interacting each target elevator controller, door controller, and presentation device. A converse approach involves the mobile computer automatically generating a user-interface based on the functional description of a target device 'on the fly'. In this approach, the wheel chair's owner does not need to pre-install any user-interface code that is specific to elevator controllers or any other device that will later be of interest. The mobile computer simply needs be able to access a possibly local user-interface generator.

User-interface generation offers several advantages over the manual approach such as low programming cost, uniformity, and as the example above shows, the flexibility to interact with devices for which code has not been pre-loaded. Therefore there has been much work on this idea [2-7,9,12,13] However, it can take a long time to dynamically create a user-interface on a mobile computer, in fact, several times longer than in the manual approach. Using a generator developed by them, Dewan and Omojokun compared local generation with preloading of local predefined user-interface code, and found that the generation times were about seven times higher and depended on device complexity [9,10]. There have been two main approaches to address this problem.

One approach, used by, Ponnekanti et al [12] and Gajos et al [3], is to use a powerful computer connected to the mobile device via a LAN to remotely generate the user-interface. To quantify the benefits of this approach, Gajos et al. measured the times needed to generate user-interfaces for a variety of devices under three configurations: (1) locally generating a user-interface on a desktop PC, (2) locally generating a user-interface on a PDA, and (3) remotely generating a user-interface on a server and rendering the user-interface on a PDA. For one of these devices, they found the times to be 1.5 seconds, 40 seconds, and 3.1 seconds respectively. Thus, they showed that remotely generating the user-interface can reduce the generation times by an order of magnitude.

The above approach requires the mobile computer be connected through a LAN to a powerful server. As the elevator example shows, this is not always possible. Moreover, in a world in which all interaction with appliances is through mobile computers, it is possible

for the server(s) to get overwhelmed with generation requests. These two problems occur because remote generation requires the use of mechanisms executing outside of a mobile computer to create a device user-interface. Hodes has proposed an alternative approach that retargets the user-interface of a source device to a target device when the source and target device are instances of the same type [4]. Under this constraint, retargeting simply involves changing the address by which (the generator code on) the mobile computer refers to the remote device. Figure 1 illustrates this approach. After turning on a light on the first floor, a user might wish to turn on a light on another floor. Rather than generating a new user-interface for the second light, the generator can simply change a reference in the mobile computer to retarget the existing user-interface to the new device.

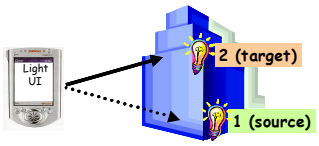

Figure 1. Retargeting a GUI between two similar lights on different floors.

Is it possible and useful to efficiently extend this concept to support retargeting among devices of different types? Let us illustrate with a simple example. Suppose that after interacting with a regular lamp, a user next interacts with a dimmable lamp, and then with a regular lamp again (Figure 2). Is it possible and desirable to efficiently morph the regular lamp user-interface to a dimmable-lamp interface and then back to a regular lamp user-interface? In this paper we answer this question for devices whose user-interfaces consist of buttons and widgets such as sliders, textboxes and combo-boxes that display values of primitive types such as integers, strings, and enums. We show that is possible to create efficient retargeting algorithms that apply to all of the existing user-interface generators known to us. Using the implementation of an existing generator and a set of real-world scenarios, we show that these algorithms can yield, in many cases, user-interface creation times that are often as good as or noticeably better than those supported by the approach of locally loading pre-existing manually-written user-interface code

The rest of this paper is organized as follows. In Section 2, we describe an abstract model of user-interface generators. In Section 3, we motivate and describe retargeting algorithms based on this model. We evaluate these algorithms in Section 4. Finally, Section 5 describes our conclusions and future work.

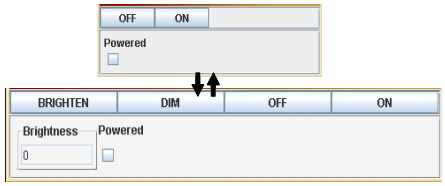

Figure 2. Retargeting back and forth between a dimmable and non-dimmable lamp.

2. Assumed Generator Model

In our retargeting algorithms, we assume that the user-interface generator has the following characteristics. It is provided with a list of the properties and commands of the device for which a user-interface is to be generated. A property is a component of the displayed state of the device associated with a public getter function to read its value and a public setter procedure to change its value. A command is any public function/procedure other than a getter or setter. These commands and properties may be automatically derived from the object coding the device [9,10], or they may be created manually using an external, language-independent, description [4,6,12] – our model does not distinguish between these two approaches as we do not measure the cost of creating an external description. As in previous generators of device user-interfaces, operation parameters and properties are restricted to simple types. Commands and properties can be collected into user-defined hierarchical view groups.

An operation is mapped to a button. Any parameter of the operation is collected using a dialogue box, which is not the subject of retargeting as it is not part of the persistently displayed state of the device. A property of a predefined type is mapped to one of a set of editable-widgets available for displaying and modifying values of the type. Each predefined value can be mapped to a text box displaying the textual representation of the value. In addition, a Boolean value can be mapped to a checkbox, an integer value to a slider, and an enumeration to a combo-box or radio button. We will refer to a widget displaying a value of type T as a T widget. Thus, we will refer to a textbox displaying a number (double, float, int), string, or Boolean value as a number, string and Boolean widget, respectively. This typing is important because we found that the cost of retargeting a widget depends on the type of the value it displays. A view group is mapped to a (potentially tabbed) panel containing the display of

its properties and operations. We do not support retargeting of properties of user-types; therefore, we do not include in our generator model how such properties are displayed. To the best of our knowledge, all of the published generators of device user-interfaces display only primitive values.

We refer to commands and properties as device elements. A device element or view group is associated with a special attribute that gives the widget to be used to display it. In addition, it is associated with a set of formatting attributes such as labels, containing view groups, and (relative or absolute) positions. The exact nature of these attributes is implementation defined. Retargeting must be aware of them only because it must ensure that the retargeted user-interface is the same as the generated user-interface. These attributes are encapsulated by an object of type `Format`. The generator provides two operations that make use of this type. The operation `getFormat(DeviceElement)` returns the format object associated with its argument. The operation `consistent(Format, Format)` checks if its two arguments are consistent, that is, they implement the same constraints on the display of the device element with which they are associated. These two operations are used by retargeting to ensure that the device elements shared by a source and retargeted user-interface are displayed using the same formatting constraints. In addition, we assume that the generator provides operations to bind a user-interface to a device, retrieve the commands and properties (and associated setters and getters) of a device, retrieve the widget/button used to display a device element, add the widget to the user-interface, remove it from the user-interface, and refresh it to show the current value of the property to which it is currently bound.

Based on our reading of the papers on published generators of devices, the model above is consistent with these generators. While these generators may not currently export the above operations, we assume that it is possible to do so to support our retargeting algorithms. We refer to the above assumptions as *design assumptions* – they influence the nature of our algorithms. In addition, we make certain *evaluation assumptions*, described below, which were necessary for us to perform our evaluation.

We assume that the cost of adding a command or property is independent of its formatting property. We make this simplifying assumption as formatting properties are generator dependent. Future generator-specific evaluations of the algorithms can determine if this is indeed true. Based on the generator we used, this assumption was true.

The final evaluation assumption is that the shared commands and properties are displayed in consistent ways in the generated user-interfaces. For example, in the user-interfaces of the dimmable and non-dimmable lamps shown in Figure 2, where the relative order of the off and on commands are the same in the two interfaces. Thus, even though our algorithms ensure that a source device's user-interface is not retargeted if the morphed user-interface would not be the same as the generated user-interface, our evaluation assumes that retargeting always succeeds.

In principle, retargeting also applies to manually created user-interfaces. However, the layout of such user-interfaces is often defined by static code that puts widgets in a certain order to create a desired format. Retargeting, in contrast, requires an ability to create a user-interface on the fly using the kinds of operations mentioned above. Performance-wise, this property is a liability when generating the user-interface from scratch. Retargeting tries to make it an asset.

In the following, we refer to the time required to create user-interface as the time required to deploy the user-interface. The user-interface may be deployed either by loading predefined local user-interface code, generating a new user-interface, or retargeting an existing generated user-interface.

3. Retargeting Algorithms

To retarget a GUI, the system invokes the exported operations of the generator to find the names of:

(a) Target-Only Commands (TOC): the target device commands that the source device does not share
(b) Source-Only Commands (SOC): the source device commands that the target device does not share
(c) Shared Commands (SC): the commands that the two devices share
(d) Target-Only Properties (TOP): the target device properties that the source device does not share
(e) Source-Only Properties (SOP): the source device properties that the target device does not share
(f) Shared Properties (SP): properties that the two devices share.

Using this information, it retargets the user-interface using the basic algorithm below:

```
Let TOC, SOC, SC, TOP, SOP, SP = lists corresponding to (a)-(f) above
{We will expand on how these lists are computed later.}
Let T = the target device
Let U = the source user-interface object
retarget (T,U, TOC, SOC, SC, TOP, SOP, SP) {
   U.setTarget(T); //referenced by buttons & widgets
   for each command_name (c) in TOC
      U.add(c)
   for each command_name (c) in SOC
```

```
      U.remove(c)
   for each property_name (p) in TOP
      U.add(p)
   for each property_name (p) in SOP
      U.remove(p)
   for each property_name (p) in SP {
        w = U.getWidget(p)
        U.refreshWidget(w)
   }
}
```

Basically, given a reference to a source user-interface object, target device, and the retargeting lists (*TOC*, *SOC*, *SC*, *TOP*, *SOP*, and *SP*) the method asks the generator to take the following steps: (a) change the target referenced by the buttons and property widgets of the source user-interface from the source device to the target device, (b) add buttons for commands in TOC (based on formatting properties of the command stored by the generator), (c) remove buttons for commands in SOC, (d) add widgets for properties in TOP (based, again, on the formatting attributes of the property maintained by the generator), (e) remove widgets of properties in SOP, and (f) refresh each widget in SP so that it shows the target device's corresponding property value.

Thus, the method above can expand and/or contract a user-interface to fit a target device with different commands and properties than the source device. This ability raises two related and important issues:

(1) *Fastest User-Interface Selection*: For a target device, let us assume that a generator has two or more potential source user-interfaces loaded in memory and none of them was created for a device that is the same type as the target. How should the generator select a source user-interface that can be changed to fit the target device in the least amount of time? To illustrate this issue, imagine the earlier person in wheelchair with a mobile computer that has the user-interfaces of some frequently used home devices still active. If this person wants to use a projector and such a device does not exist at home, which source user-interface should the generator pick for the 'fastest' retargeting?

(2) *Approach Selection*: How should a generator decide whether it is faster to retarget the fastest user-interface or generate a new one for the target device?

We address both issues using the novel idea of *regression-based source-device prediction*. A generator selects the fastest user-interface to retarget by using a function that estimates the retargeting time of each potential source user-interface. This function accepts the amount of work required to change a user-interface and returns a retargeting time estimate. Given an estimate for each potential source user-interface, the generator then selects the one with the lowest value. Similarly, to predict the faster of the two approaches (generate or retarget), a generator uses a function that estimates the time to create a new user-interface for the target device. This function accepts the amount of work required in creating a whole new user-interface and returns a generation time estimate. Given this generation time estimate and the retargeting time estimate for the fastest source user-interface, the generator selects the approach with the lowest value.

We derived an outline of the two estimation functions by first identifying the high-level steps (or sub-operations) involved in algorithms of the respective approaches. Our retargeting algorithm involves steps (a)-(e), as described earlier. Generation, on the other hand, involves: (a) creating an empty user-interface frame, (b) adding buttons for invoking the target device's commands, and (c) adding widgets for displaying the target device's property values. Profiling the generator shows that the time it takes to remap, remove, and add a property widget depends on the type of widget. For example, the time it takes to create and add a string widget to a user-interface is more than the time it takes to perform the same operation on a boolean widget. We found three kinds of widgets with significant sub-operation time differences: numeric (int, float, double, and long), string, and boolean widgets. Thus appropriately, each approach's estimation function is the following sums:

{Let: *BA*=# buttons to add; *BD*=# buttons to remove; *BR*=# buttons to remap; *NWA*=# num widgets to add; *BWA*=# bool widgets to add; *SWA*= # string widgets to add; *NWD*=# num widgets to delete; *BWD*=# bool widgets to delete; *SWD*=# string widgets to delete; *NWR*=#num widgets to remap; *BWR*=# bool widgets to remap; *SWR*=# string widgets to remap}

1) $T_{ret}(BA, BD, BR, NWA, BWA, SWA, NWD, BWD, SWD, NWR, BWR, SWR) = T_{add\_btn}(BA) + T_{rmv\_btn}(BD) + T_{rmp\_btn}(BR) + T_{add\_num\_wdgt}(NWA) + T_{rmv\_num\_wdgt}(NWD) + T_{rmp\_num\_wdgt}(NWR) + T_{add\_bool\_wdgt}(BWA) + T_{rmv\_bool\_wdgt}(BWD) + T_{rmp\_bool\_wdgt}(BWR) + T_{add\_str\_wdgt}(SWA) + T_{rmv\_str\_wdgt}(SWD) + T_{rmp\_str\_wdgt}(SWR)$

{Let: *BG*=# buttons to generate; *NWG*=# num widgets to generate; *BWG*=# bool widgets to generate; *SWG*=# string widgets to generate}

2) $T_{gen}(BG, NWG, BWG, SWG) = T_{gen\_frm} + T_{gen\_btn}(BG) + T_{num\_pwdgt}(NWG) + T_{bool\_pwdgt}(BWG) + T_{str\_pwdgt}(SWG)$

T_{ret} estimates retargeting time by summing the results of functions that estimate the completion times of the retargeting sub-operations based on given workload values. The subscript and parameter of each sub-operation's function describe the process the function estimates. For instance, $T_{add\_btn}(BA)$ estimates the cost for adding *BA* buttons.

T_{gen} estimates generation time by summing the results of functions that estimate the completion times of the generation sub-operations based on given

workload values. T_{gen_frm}, in T_{gen}, has no parameters because generating an empty frame is a static operation—it should therefore have a constant value.

Given the outlines for the two time estimation functions, we defined the actual calculations involved within the sub-operation functions. We achieved this goal by using regression, which derives an empirical function from a set of experimental data. To gather the necessary data, we used timestamps to measure actual times for performing the various sub-operations over a range of workloads. We, for example, measured:

$$T_{add\_btn}(1), T_{add\_btn}(2), \ldots T_{add\_btn}(N), N=42$$

For each sub-operation, *N* represents an integer that is at least the maximum input value to which the generator will be exposed during interaction. We looked at all the cases of retargeting and generating user-interfaces of our six experimental devices (identified later) to pick the *N* values. The value we chose for each sub-operation was at least the maximum value possible. Ideally, a very large *N* value would be chosen to support arbitrary devices.

In our evaluations, we performed profiling experiments on a pocket PC and two laptops: an iPAQ (206MHz StrongArm, 32MB), a "slow" 400Mhz Celeron - (64MB), and a "fast" 733 MHz Pentium (128MB). The fast laptop yields times that are significantly lower than the slow laptop and iPAQ. For instance, the slow laptop takes approximately seven times longer than the fast laptop to add forty-two buttons to an empty user-interface. Also, it can take nearly five times longer to remap a property widget on the iPAQ than on the slow laptop. Thus, these times are very sensitive to the computer platform.

Using the collected results, we applied MATLAB's polynomial fitting command (called `polyfit`) on each sub-operation's data set to find its empirical time-cost function. Given the linear behavior of the data, we chose a degree of one for each function. Below is an example, which shows fast laptop's T_{ret} and T_{gen}:

1) $T_{ret}(BA, NWA, BWA, SWA, NWD, BWD, SWD, NWR, BWR, SWR) = 0.16BA + 6.17NWA + 3.63BWA + 5.67SWA + 3.37NWD + 2.94BWD + 3.28SWD + 0.31NWR + 0.36BWR + 0.50SWR - 35.22$

2) $T_{gen}(BG, NWG, BWG, SWG) = 2.70BG + 6.17NWG + 3.63BWG + 5.67SWG + 22.24$

The entire process that leads to the two functions, for a given mobile computer, could be automated by the notion of a *self-profiling generator*. Such a generator would run a bootstrap program that automatically performs all the necessary profiling operations and measures its own performance on the computer. The program would then run regression code to return an appropriate T_{ret} and T_{gen} function.

As implied by the ten parameters required by the above T_{ret} function, a problem of *regression-based source-device prediction* is long search times. In order to predict the fastest user-interface to retarget, a generator must search through the commands and properties of the source and target devices to determine T_{ret}'s parameter values. If the target and source devices are complex or there are many potential source user-interfaces, searching can become expensive, as we found in our experiments. For this reason, we support a complex cache-based retargeting scheme.

A generator caches the T_{ret} value it calculates for each source and target device type pair it ever evaluates. If a pair with a cached T_{ret} value occurs again in the system, the generator avoids recalculating T_{ret}, which involves finding values of the ten parameters of the functions.. Instead, it simply retrieves the stored value. Notice the similarity between the parameter values required by T_{ret} and the retarget lists (*TOC*, *SOC*, *SC*, *TOP*, *SOP*, and *SP*) for `retarget()`. T_{ret} requires the number of buttons and state widgets that must be added, removed, and remapped to change a source user-interface to fit the target device. The `retarget()` method requires lists that contain the names of commands and properties that correlate to these same buttons and state widgets that must be added, removed, and remapped. In determining T_{ret}'s parameter values, the algorithm inherently builds the lists containing these names. That is, the lists are a byproduct of gathering T_{ret}'s parameters. These lists are thus cached so that they can be accessed and passed to `retarget()` if the generator decides to retarget.

On occasions where there are multiple potential source user-interfaces to retarget to a target device, the generator also caches the corresponding source device type of user-interface that it predicts to be the fastest. If a user wants to use a device of the target's type later and the same set of source user-interfaces is available, the system directly picks the user-interface associated with the cached device type. Thus, it avoids all the operations involved in finding the fastest user-interface. The system also caches T_{gen} values for each target device type to avoid repeating the process of gathering the function's parameters. The following describes this process more formally:

Let *C1* = a cache. For a given set of source device types (*S*) and target device type (*t*), *C1* caches: (1) the decided source device type ($s_{fastest}$) in *S* with the lowest T_{ret} value (T^{min}_{ret}) and (2) the associated T^{min}_{ret} value. Imagine *C1* as a hash table: *C1*.put([*S*,*t*], [$s_{fastest}$, T^{min}_{ret}]) inserts the elements in the cache and *C1*.get([*S*,*t*]) returns the cached collection [$s_{fastest}$, T^{min}_{ret}]

Let *C2* = a cache. For a given source device type (*s*) and target device type (*t*), *C2* stores the

```
results from calculating T_ret and 'retargeting lists'
{TOC, SOC, SC, TOP, SOP, SP}.  Imagine C2 as a hash
table: C2.put([s,t], [T_ret , TOC, SOC, SC, TOP, SOP,
SP] ) inserts the elements in the cache and
C2.get([s,t]) returns the cached collection [T_ret ,
TOC, SOC, SC, TOP, SOP, SP]
Let C3 = a cache.  For a given target device type
(t), C3 stores the type's T_gen value.  Imagine C3 as a
hash table: C3.put( t, T_gen ) inserts the elements in
the cache and C3.get(t) returns T_gen.
Let S = the set of device types of the currently
  available source UIs
Let U = the set of source user-interface objects
Let t = the target device

boolean retargetHomogeneous(U,t) {
   match = getMatchingSourceUIforType(U,t);
   if (match != null)  {
      retarget(target, match, null, null,
               getCommandNames(t), null,
               null, getPropertyNames(t))
      return true
   }
   else
      return false
}
boolean retargetHeterogeneous(C1, C2, C3, S, U, t) {
   [s_fastest , T^min_ret ] = C1.get([S,t])
    if  ([s_fastest , T^min_ret ] == null) {
       for each source device type (s) in S {
          if retargetingDoesNotYieldGeneratedUI(s,t)
                // use consistent operation
                //provided by generator
            continue;
            [T_ret,TOC,SOC,SC,TOP,SOP,SP]=C2.get([s,t])
             if ([T_ret,TOC,SOC,SC,TOP,SOP,SP]==null) {
                [T_ret , TOC, SOC, SC, TOP, SOP, SP] =
                     computeTret(s,t)
                C2.put([s,t],[T_ret,TOC,SOC,SC,TOP,
                           SOP,SP])
             }
             [s_fastest,T^min_ret]=min(s_fastest,T^min_ret,s,T_ret)
       }
       C1.put([S,t],[s_fastest,T^min_ret])
    }
    T_gen = C3.get(t)
    if (T_gen == null) {
       T_gen=computeTgen(t);
       C3.put(t,T_gen )
    }
    if (T^min_ret <= T_gen)  {
      [TOC,SOC,SC,TOP,SOP,SP] =
         getRetargetingLists(C2.get([s_fastest ,t]))
      retarget(target,getMatchingSourceUIforType(U,
           s_fastest ),TOC,SOC,SC,TOP,SOP,SP)
      return true
    }
    else
   return false
}
retargetORgenerate(C1, C2, C3, S, U, target) {
   t = getType(target)
   if (retargetHomogeneous(U,t) )
      return
   else {
      if retargetHeterogeneous(C1,C2,C3,S,U,t)
        return
      else
        generateUI(target)
   }
}
```

The `retargetORgenerate()` method accepts the references to: cache *C1*, cache *C2*, cache *C3*, the set of source device types (*S*), the set of source user-interface objects (*U*), and the target device (*target*). It assumes that the fastest user-interface to retarget is always the one created from a source device that is the same type as the target device (*t*). Thus, it first checks for such a user-interface by first calling `retargetHomogeneous()`. Further, this method assumes that retargeting this user-interface is always faster than generating a new one because retargeting would only involve remapping user-interface components.

With *t*, `retargetHomogeneous()` calls `getMatchingSourceUIforType(U,t)`, which searches the set of source user-interface objects to see whether one has already been created for a source device of type *t*. If such a user-interface object exists, `retargetHomogeneous()` calls `retarget()` to actually retarget the object. It then returns true, notifying `retargetORgenerate()` that it performed the retargeting. Notice the null values passed into `retarget()` for *TOC*, *SOC*, *TOP*, and *SOP*. The reason for them is that when retargeting a user-interface between two devices of the same type there are no buttons and property widgets to add and remove. All components are shared.

If `getMatchingSourceUIforType(U,t)` does not return a matching user-interface, then `retargetHomogeneous()` returns false. The result of `retargetHomogeneous()` decides the next step in `retargetORgenerate()`. With a true result, `retargetORgenerate()` terminates since `retargetHomogeneous()` completed the actual retargeting. Otherwise, it must decide whether to: (1) retarget the fastest source user-interface created for a device of a different type than the target or (2) generate a new one.

To decide on which approach to take, `retargetORgenerate()` calls `retargetHeterogeneous()`, which accepts *C1*, *C2*, *C3*, *S*,*U*,and *t*. The first step of `retargetHeterogeneous()` is to check the cache *C1* to see if the specific set *S* and type *t* have been previously evaluated to find the source device type ($s_{fastest}$) that yields the lowest $\boldsymbol{T_{ret}}$ value (T^{min}_{ret}). If so, it stores the $\mathbf{s}_{fastest}$ and T^{min}_{ret} value from the cache in a variable. Otherwise, the method begins searching for $\mathbf{s}_{fastest}$ and T^{min}_{ret}. This involves getting the $\boldsymbol{T_{ret}}$ value for each source and target device type pair (*s*,*t*) produced by *S* and *t*. The pair with the lowest $\boldsymbol{T_{ret}}$ value (T^{min}_{ret}) contains $\mathbf{s}_{fastest}$. It finds these values by first checking the cache *C2* to see if a given pair has been previously evaluated to find its $\boldsymbol{T_{ret}}$ value and the corresponding retargeting lists *TOC*, *SOC*, *SC*, *TOP*, *SOP*, and *SP*. If so, then it stores this collection in a variable. Otherwise, it must call `computeTret()` to determine these values. Given *s* and *t* this method searches their programming interfaces to determine the parameter values needed for $\boldsymbol{T_{ret}}$() and then calculates the function's value. Recall that the retargeting lists for

the pair is a byproduct of this process and is thus returned with T_{ret}. Thus, `computeTret()` returns a collection consisting of T_{ret} and the retargeting lists. This returned collection is inserted into cache *C2*. As `retargetHeterogeneous()` evaluates each pair (*s*,*t*), it uses `min()` to keep track of the $s_{fastest}$ it is has seen so far with the lowest T_{ret} value (T^{min}_{ret}). After it is done evaluating each pair (i.e. the loop), the final $s_{fastest}$ and T^{min}_{ret} are appropriately defined for (*S*,*t*). It inserts this information in the cache *C1*.

With $\mathbf{s}_{fastest}$ and the corresponding T_{ret} value decided for (*S*,*t*), `retargetHeterogeneous()` moves on to decide whether to retarget or generate. It checks cache *C3* to see whether T_{gen} for the target device's type (*t*) has ever been calculated. If so, then it stores this value. Otherwise, it must search the description of the commands and properties of type *t* to get T_{gen}()'s needed parameter values. It then calculates the function's value. At this point, the method knows $\mathbf{s}_{fastest}$, T_{ret}, and T_{gen}. If T_{ret} is lower or equal to T_{gen}, it executes `retarget()`, passing in the reference to the target device, source user-interface generated for $\mathbf{s}_{fastest}$, and retargeting lists. It then returns true, notifying `retargetORgenerate()` that it retargeted. Otherwise, it generates a new user-interface for the target device.

4. Evaluation

In this section, we evaluate our retargeting algorithms given then above. We performed many experiments for this evaluation. However, due to limited space, we highlight the most important results in this paper.

Our evaluation focused on several questions:

1) *Source User-Interface Selection Performance* - In scenarios where multiple source user-interfaces are available for retargeting, is the one with the lowest T_{ret} value actually the fastest?
2) *Approach Selection Performance* - Does picking the lower value between T_{ret} and T_{gen} accurately decide the faster approach—retarget or generate?
3) *Retargeting Performance* - Can retargeting allow a generator to be competitive with loading of predefined manual user-interfaces?
4) *Cache Performance*: To what extent does our caching scheme affect the performance?
5) *Device complexity*: How does device complexity (in terms of number of commands and properties) influence the answers above?
6) Network connectivity: How does the speed of the network between the mobile computer and the networked device influence the answers above?

Naturally the answers to these questions depend not only on the algorithms but also several other important factors – the generator used; the set of devices considered; and the tasks for which these devices are used, which determine what source user-interfaces are available when a user-interface for a new device must be deployed. We have reduced the influence of the computing platform by asking questions about relative rather than absolute performance. We performed experiments on two kinds of mobile computers: a laptop and an iPAQ. Because of issues with the stability of our Java-based iPAQ platform, our laptop experiments are more complete. Therefore, we primarily report them instead of the iPAQ ones.

For the generator, we used one that implemented all of the design and evaluation assumptions of the generator model described earlier. Our results should apply to other generators that follow these assumptions. As mentioned earlier, we believe that current generators do satisfy the design assumptions, but future work is required to determine if they also satisfy the evaluation assumptions. To answer the device complexity question, for each device, we created a command-only user-interface and a command-and-state based user-interface. The former arranged a single view group containing buttons for all of the commands, and the latter, in addition, created a view group displaying the properties. The exact format of the user-interface does not matter because, as mentioned above, we assume that (a) generators arrange common commands and properties in a consistent fashion in different user-interfaces, so retargeting always works, and (b) the cost of displaying a command or property is independent of its formatting attributes. To compare the cost of deploying retargeted user-interfaces with loading predefined user-interface code, for each device, we handcrafted a command-only and command-and-state user-interface that used the same widgets as the corresponding generated user-interface. In all the user-interfaces, we used Java Swing widgets.

For devices, we considered six appliances to which the first author had access: a TV, VCR, A/V Receiver, DVD Player, Projector, and lamp. For each of these devices, we created a networked Java proxy object that represented a network version of the device. The object offered all of the commands supported by their remote controls. In addition, it supported primitive properties such as the 'Powered' property in Figure 2 that could be derived from these commands. The mobile computer interacted with the real device through the networked proxy object.

For user-tasks, we wanted to use real world sequences of device accesses. Thus, we collected

interaction data from different users performing their device-related tasks. We gathered the data in two kinds of environments where people frequently use devices—at their respective homes and a conference room. To gather such data, we built a tool that records the IR-based interactions emitted by remote controls. We logged 11 individual participants at their homes for a period varying from one to two weeks, producing a total of well over 30,000 recorded commands. In selecting the participants, we only considered the people who owned the types of IR devices that we networked (TV, lamps, DVD player, VCR, and A/V receiver). This criterion maximized the kinds of experiments we could run using our six devices.

The conference room, on the other hand, is a static environment and consists of a projector and three lamp arrays. We were especially interested in the task of 'setting up for a presentation', which involves a series of deterministic device accesses. Thus, we did not need to use our IR recording mechanism in this room. Essentially, the task involves turning on the lamps in the room, setting up the projector, and then dimming (or turning off) the lamps.

4.1. Source User-Interface Selection Performance

Our first step in evaluating T_{ret}'s prediction performance was to identify a benchmark set of scenarios. We were able to use our participants' logs to produce this set. From interviewing the participants, we could determine when a participant goes from one task directly to another. Assuming the participants had mobile computers that could perform retargeting, the user-interfaces from the previous task could immediately be available as sources for the next. To illustrate, imagine a person who watches cable TV for a while and then watches a DVD. The TV or the cable box user-interfaces would serve as a source to retarget to the DVD player; i.e., the mapping: (TV UI, cable box UI → DVD player).

We found three unique transitions within the logs, with many of the participants producing the same cases. Using the presentation room example, we supplemented these examples with two more. In particular, we imagined a user who enters a conference room with the user-interfaces of four commonly used home devices (a TV, VCR, Receiver, and lamp) still running on a client. The four user-interfaces are thus available for retargeting to the projector and lamps in the room. The first two columns of Table 1 summarize our five transitions.

For each identified transition, we evaluated the source user-interface selection performance when deploying the command-only and command-and-state based GUIs. The experiments show that that prediction using T_{ret} correctly picks the fastest user-interface for all five cases regardless of the kind of user-interface being deployed. Further, the prediction is successful even when the difference between the times that two potential source user-interfaces offer is only one percent. The results also show the importance of selecting the source user-interface that is actually the fastest. To illustrate, for the 'turning on the conference room lights' task transition, there is a 95% increase in retargeting time when choosing the receiver's user-interface over the lamp's.

4.2. Approach Selection Performance

The results show that selection using the lower value of T_{ret} and T_{gen} correctly picks the fastest approach for all transitions when deploying command-only and command-and-state based GUIs. In fact, retargeting is always at least twice as fast as generation, even when the source and target are very different devices. Of course, with a more diverse device set, the answer may be different.

4.3. Retargeting Performance

Using our real world scenarios, we evaluated how close two important levels retargeting (homogeneous and heterogeneous) can achieve times that are comparable to locally loading the pre-defined user-interface code we created for the six devices.

For our homogeneous retargeting evaluation, we assume that a source device that is the same type as the target device is always available. In other words, it expects that (in memory) there is a source user-interface object previously made for a device that is of the target device's exact type. Heterogeneous retargeting avoids this assumption by supporting different device types. Some user-interface simply needs to be in memory.

Table 1 compares locally loading pre-installed code to the two kinds of retargeting on the laptop for command-only and command-and-state based GUIs when the mobile computer and the device it controls are on a high-speed wired LAN. It shows five important points:

(1) Regardless of the kind of user-interface deployed (command or command-and-state based), homogeneous retargeting times are an order of magnitude lower than their corresponding times of locally loading predefined code.

Table 1. The five identified transitions and associated deployment times. The homogenous column shows resulting times if homogeous retargeting was possible. {*The DVD player also serves as a music CD player. The underlined device marks the one selected as the fastest to retarget }

Task Transition	Mapping (source UIs → target device)	Times (ms)						
		Command-Only			Command and State			
		Local load	Retarget		Local load	Retarget		
			Homo-geneous	Hetero-geneous		Homo-geneous	Heterog-eneous	
							N-C	C
Watch TV after a DVD movie	TV,DVD,RCVR –-> VCR	255.50	6.25	241.42	297.13	24.44	280.67	155.78
Watch a DVD movie or listen to music after watching TV*	TV, RCVR –-> DVD	250.67	10.00	277.50	270.50	40.00	409.44	277.11
Watch a DVD movie or listen to music after watching TV*	TV,VCR,RCVR –-> DVD	250.67	10.00	200.00	270.50	40.00	396.00	266.00
Turn the lights on in a conference room	TV,VCR,RCVR,LAMP–-> LAMP	123.88	5.71	5.71	151.83	11.11	11.11	11.11
Setup a projector in a conference room	TV,VCR,RCVR,LAMP–-> PROJ	219.00	10.00	197.88	232.88	19.00	260.44	111.33

For heterogeneous retargeting, the times depend on whether: (a) the source user-interface is state-based and (b) caching is turned on.

(2) With no caching and command–only user-interfaces, retargeting mostly offers lower times than the times of locally loading pre-defined code.

(3) For command-and-state based user-interfaces, however, nearly all of the non-caching based retargeting times are significantly above their corresponding times of locally loading predefined code. Only one of the five cases, 'TV*,DVD,RCVR–>VCR', yields a retargeting time that is lower than its corresponding locally loading based time. In this case, the TV GUI is picked for retargeting. The TV and VCR are our two most similar devices in terms of the commands and properties that they offer—hence this exception.

(4) Turning on caching significantly improves heterogeneous retargeting time. On the laptop, most of the retargeting cases are faster than locally loading predefined code after activating caching.

Due to lack of space, we omit images of the user-interfaces deployed in our experiments. See [9] for these user-interfaces.

The absolute values of the deployment times are small, and in fact, in many cases, not noticeable to users. However, as previous work has shown [1,9], significantly longer wait times can occur under one or more of the following circumstances: (a) using a much less powerful computer than the laptop (e.g. a PocketPC), (b) using a slower connection than the wired LAN, and (c) deploying speech UIs [9,10]. Our preliminary results using our Java-based iPAQ and a wired LAN connection, for example, show that cache-based retargeting can cut generation time by multiple seconds. We do not report the iPAQ results because stability issues did not allow us to carry out many of our experiments to completion. Therefore, what is important in the table is the relative rather than absolute performance. Part of our future work, discussed later, involves exploring absolute performance of retargeting in the above circumstances.

The designers of SUPPLE have addressed the multiple-second wait times in these circumstances by caching user-interfaces, which works only when a user-interface is reused [2,3]. Our retargeting approach is an extension of caching that supports efficient transitions to different user-interfaces.

5. Conclusions and Future Work

Generating a user-interface on the fly has many advantages but, in comparison to loading local predefined user-interface code, significantly increases the time it takes to deploy the user-interface of a device on a mobile computer. Retargeting makes this property an asset rather than a liability. This basic idea was invented by Hodes for homogeneous devices [4]. This work, however, offered no performance evaluation. Moreover, it did not address the much more complex problem of retargeting heterogeneous devices. This paper fills these two gaps within pervasive computing.

Our most abstract message is that retargeting is an important research direction because it can significantly reduce deployment times. Homogeneous retargeting can in fact yield times that are much lower than the cost of loading predefined user-interface code. Not surprisingly, heterogeneous retargeting is more flexible but yields a lesser benefit. What is surprising is that, for the devices and scenarios we studied, it yielded deployment times that were at least twice as fast as generation times, and often as good as the times offered by loading predefined local user-interface code.

Our next-level evaluation message is that the benefits of retargeting depend on the complexity of the device and the network speed between the mobile computer and device. In particular, the benefits are lower (a) for command and state user-interfaces in comparison to command-only user-interfaces, and (b) dialup rather than wired connections between the device and mobile computer.

Our work also contributes to the design of retargetable user-interface generators. At the most abstract level, it shows that such a generator must (a) morph a source user-interface to create a target user-interface, (b) perform a profiling process on each platform that measures the time required to add and delete widgets of different types to the user-interface, (c) use the profiling results to predict the cost of generating and retargeting user-interfaces, and (d) use a cache to store the data gathered in previous retargetings. Our next-level design contributions are the specific algorithms for implementing these tasks.

Of course, with a different generator, set of devices, and usage scenarios, the evaluation results may be different. This paper, thus, motivates more experimental work on retargeting. In particular, it is important to determine if the evaluation assumptions are valid for other generators. Moreover, it would be useful to extend (a) the set of target devices in the domain examined by including, for example, thermostats, sensors, and door controls, and (b) the mobile computers used in the experiments by including, for instance, cell phones.

Similarly, the retargeting algorithms given above are simply a first-cut effort in this area. In particular, it would be attractive to predict whether retargeting offers better performance than remote generation and choose the better approach when both approaches are available. Also, it would be useful to explore how the algorithms above can be adapted to support retargeting of generated speech-based user-interfaces, which are even slower to generate than graphical ones. On the "fast" laptop described here, Omojokun and Dewan's speech-based generator [9,10] can take many seconds to generate a receiver's user-interface. Given the significant delay on this relatively powerful machine, it could also be attractive to explore the idea of remote generation and retargeting, and thus marry the ideas of remote servers, generation, and retargeting. Such a marriage would be more attractive in the future, given the increased pervasiveness of WiFi hotspots as well as emerging 4G infrastructure such as WiMax. Yet another intriguing extension is to consider retargeting of compositions (or “mash-ups”) of multiple devices [7,11,12]. Finally, retargeting a user-interface of a device to user-interfaces of different kinds of devices is similar to the idea in [5] of adapting a user-interface of an device to different styles/preferences. It would be useful to explore, in depth, the relationship and possibly integration of these two seemingly related ideas.

5. Acknowledgements

This research was funded in part by IBM, *Microsoft* and NSF grants ANI 0229998, EIA 03-03590, IIS 0312328, and IIS 0712794.

6. References

[1] Edwards, W., et al. *Recombinant Computing and the Speakeasy Approach*. In *Mobicom 2002*.

[2] Gajos, K. and D.S. Weld. *SUPPLE: Automatically Generating User Interfaces*. In *IUI 2004*.

[3] Gajos, K., et al. *Fast And Robust Interface Generation for Ubiquitous Applications*. In *UBICOMP 2005*.

[4] Hodes, T. and R. Katz, *Composable Ad Hoc Location-Based Services For Heterogeneous Mobile Clients.* Wireless Networks, 1999. **5**: p. 411-427.

[5] Nichols, J., B.A. Myers, and B. Rothrock. *UNIFORM: Automatically Generating Consistent Remote Control User Interfaces*. In *CHI 2006*.

[6] Nichols, J., et al. *Generating Remote Control Interfaces for Complex Appliances*. In *UIST 20*02.

[7] Nichols, J., et al. *Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances*. In *UIST 2006*.

[8] Nylander, S. and Bylund, M. *The Ubiquitous Interactor: Universal Access to Mobile Services.* In *HCII 2003*.

[9] Omojokun, O. and P. Dewan. *Automatic Generation of Device User-Interfaces?* In *PerCom 2007*.

[10] Omojokun, O. and P. Dewan. *Experiments with Mobile Computing Middleware for Deploying Appliance UIs.* In *ICDCS 2003–Workshops*.

[11] Omojokun, O. and P. Dewan. *A High-level and Flexible Framework for Dynamically Composing Networked Devices*. in *Proceeding of 5th IEEE WMCSA* 2003.

[12] Ponnekanti, S.R., et al. *ICrafter: A Service Framework for Ubiquitous Computing Environments*. In *UBICOMP ‘01*.

[13] Reitter, D., E. Panttaja, and F. Cummins. *UI on the fly: Generating a multimodal user interface.* In *HLT/NAACL 04*.