Virtualized Approach Towards Achieving Seamless Mobility

Dushmanta Mohapatra , Umakishore Ramachandran
Georgia Institute of Technology
{*dmpatra,rama*}*@cc.gatech.edu*

Xiang Song
Microsoft Corporation
songxiang@gmail.com

Jatin Kumar
nVIDIA Corporation
jatinkumar@gmail.com

Sang Bum Suh
Samsung Electronics
sbuk.suh@samsung.com

Abstract

The advent of the genre of personal computing and its gradual evolution into the 'state of the art' pervasive and ubiquitous computing era has resulted in the general user having access to a multitude of computing devices (PCs, Laptops, PDAs and cell phones). And with the increase in computing and storage capabilities of small hand-held devices, people have started using them for internet and multimedia related services. Add to that the increasing mobility of users and the difference in resource availability of different devices; and there is a need to provide mechanisms for service migration from one machine to another.

However the heterogeneity and varying capability of I/O components (in different devices), makes seamless service mobility very difficult to achieve. There is a need for service customization in accordance with the changing environment. This could happen at the application layer or (in case the application does not support) it could happen inside operating system. In this paper we present an integrated approach towards achieving seamless mobility, using service state virtualization at the application layer and device state virtualization and capability adaptation inside operating system. We use Xen VMM for device interface virtualization and have devised a mechanism for inducing situation dependent capability adaptors into the IO data path

1 Introduction

Many visions of the future predict a world with pervasive computing, where computing services and resources permeate the environment. In these visions, people will want to execute a service on any available device without worrying about whether the service has been tailored for the device. As a concrete realization of this vision, people nowadays tend to use their mobile devices to enable them to work at any place at any time. However, mobile devices are usually constrained by their limited power, storage, display etc. But with the advancement of technology, it is becoming increasingly feasible for users carrying mobile devices to take advantage of nearby resources in the environment to enhance their experience [2].

In addition to using the environment provided resources, mobile users may also want to seamlessly migrate their work from place to place. A person watching a movie at the lounge, and unable to not finish it before boarding, might want to migrate his movie (some how) to the small monitor in front of his seat on the plane and to continue it without worrying about where he left off.

However the difficulties in achieving the above technicalities are many. It is getting increasingly difficult to create services that can execute well on the wide variety of devices being developed at present, primarily because of problems with diversity and resource constraints. Another important issue for such migrations is that the IO devices at source and destination might be differing in their internal configurations. Output devices may have different sizes/resolutions, color settings. Input devices may also be different in the keyboard mapping, mouse accuracy or keypad functions. A good migration system has to take these heterogeneities into consideration and thus there is need for a mechanism to adapt according to these changing IO device characteristics.

Some research projects address this problem by application or middleware level adaptation. The main idea of these approaches is that the applications know the configuration of physical devices and can tune the output according to the configuration[8, 5, 4]. However,a drawback of these approaches is that it requires the application developer to understand physical device specifications and make necessary changes in the output with that information. Therefore, legacy applications will not work without being modified and recompiled. Further complications arise when the application developer does not have access to device specification or cannot adjust

the output accordingly.

In this paper, we present an integrated architecture for supporting seamless mobility. We have support for application/service level adaptation. This is used when there are applications available in source and destination machines to provide the intended service. The additional support needed is a mechanism for proper (service)state transfer. But there will be situations where this approach is not feasible. Fortunately system virtualization [3] and virtual machine (VM) migration [7] techniques could be used in such cases to migrate an entire VM(guest) into the destination machine (which will be the host). We use Xen VMM based virtualization and VM migration functionalities to put together an interface virtualization system, called Chameleon, that can dynamically adapt to bridge the gap between the IO device configurations of the guest and host. In our system architecture, adaptation happens inside the operating system. There is no additional requirement for the application to be aware of lower layer details. This transparent capability adaptation mechanism releases the application developer from dealing with device-specific output adaptation support and hence allows legacy applications to be migrated to the destination without modification.Capability adaptation is achieved by allowing dynamic installation and uninstallation of capability adaptors (the modules responsible for adaptation).

Although the system architecture presented in this paper is generic and could be applied for a variety of services and IO devices, as a proof of concept implementation we have focussed on movie playing service and adapting to varying display device characteristics and configurations. The development of sophisticated capability adaptor algorithms is beyond the scope of this project. There is a lot of work done by HCI researchers in this domain [13, 18], which is orthogonal to our work and can be complementary to our system for better user experience.

The rest of the paper is organized as follows: after presenting our design principles in section 2, we briefly describe about Xen and why we choose Xen to implement our framework in section 3. Then, in section 4 we present our design, followed by some details of implementation in section 5 and the performance evaluation in section 6. Finally, after briefly summarizing other related works, we conclude our paper and describe possible future work.

2 Design Principles

As discussed in section 1, the primary intent of our research effort has been directed towards creating a system for achieving service migration as seamleslly as possible. The number of ways in which a solution to this issue could be devised is also quite a few is definitely quite a few. So in order to keep us on track, we formulated a set of design principles.

Before discussing the design principles, we want to describe the system briefly. One type of migration is service state migration from an application in the source machine to an application in the destination machine (the source and destination applications might be same or different). The other type of migration is whole OS(/Virtual Machine). The first design principle is that the migration form most appropriate to the situation will be used.

In case of the OS(/VM) migration, there is a mobile platform that is moved with the user and a stationary host system (environment) available in each machine. So following the terminologies used in paravirtualization world, the mobile platform is termed as "guest" system. The guest system does not interact directly with the machine hardware. It uses the "host" system functionalities for this purpose, and thus may be migrated to a different host system if the user moves.

The second design principle is that we want to keep the mobile platform unchanged and make the environment adapt to the mobile platform. With VM migration there is always a question of where the adaptation should happen. It could happen either inside the guest system or inside the host system. In a typical virtualized computing environment, there are tens of guest OSes and one host OS on which the guest OSes depend. The host OS always knows the local resources better and hence doing the adaptation inside the host might be easier and more effective. Another way is to make the migrating guest domain adapt to host environment configurations. This approach might not work well in practice because it puts the burden of learning about the environmental configurations on the guest machine, which will be hard to achieve given its migrating nature and the increasing heterogeneity of our computing infrastructure.

The third design principle is that we want to build a system that can support the dynamic installation and uninstallation of capability adaptor modules for adaptation. The adaptation algorithms, however, are not our main concern and are out of the scope of our research. We rely on other researchers (probably in the field of human computer interaction) to develop the algorithms to be used for adaptation and focus on providing an easy way for their inclusion in the overall system.

The fourth design principle is that we want to make the selection of adaptation algorithms automatic and at run-time.

The fifth design principle is that we want to make the migration as seamlessly as possible. Therefore, we may need to migrate the device states (such as frame buffer contents for display devices) in addition to capability adaptation. By dumping and resuming the device

states, we can ensure the seamless migration of the domain without losing any states that are out of the control of the software system.

3 Background

3.1 Xen and Virtual Device Framework

In order to keep in synch with the design principles described in the previous section, we chose Xen and Xeno-Linux(a patched linux system to work with Xen) as the base of our system architecture. In this section, we briefly describe about some necessary features of Xen based virtualization that have helped us in our research.

Xen is an open-source virtualization system that can support multiple guest operating system running on top of Xen hypervisor, sharing the same hardware resources. In the Xen paravirtualization approach, there is a host operating system (Xeno-Linux) running on top of the hypervisor which manages the hardware resources and acts as a control point for the guest operating systems. By virtualizing the hardware resources, the hypervisor gives the guest operating systems the illusion of having control over the resources exposed to them, while internally it manages the sharing of the resources across different guests. In addition, the hypervisor creates different domains for these operating systems in separate address spaces, called virtual machines (VM), and Xen provides facilities to suspend,resume and migrate these VMs.

Split device driver model is a key feature in Xen that makes the capability adaption possible without significant change in existing Xen virtualization setup. Figure 1 shows the basic architecture of Xen system. The host operating system contains the backend device drivers (BE) similar to physical device drivers in traditional operating systems. These backend drivers are device specific and have direct control over the physical devices. The guest operating system(s) contain generic frontend device drivers (FE) which do not have direct access to the physical devices. Frontend drivers connect to backend drivers when the guest OS is loaded into the system. After a complex handshaking process and con- nection establishment, FE can forward the application requests of accessing the device (such as reading or writing) to BE. BE then processes those requests to fulfill FEs needs, just like traditional device drivers do.

In such a split device driver model, the frontend drivers can be considered as virtual device drivers that are generic and do not deal with device specific details. The virtual device driver becomes concrete when it gets connected to a backend driver, which has device specific information (e.g. resolution,color settings etc.). Applications running in the guest operating system interact only with virtual device drivers.

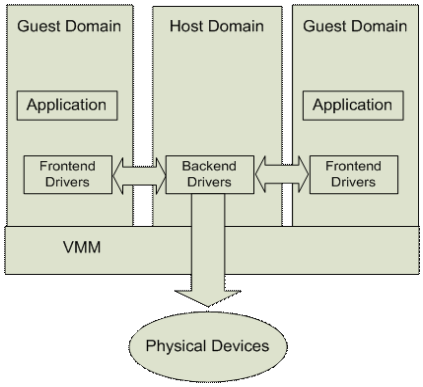


Figure 1: Xen Architecture

Based on the above discussion, the key features that helped us develop an interface virtualization and capability adaptation system are:

- Xen has generic frontend device drivers, which provided us with a very good starting point to develop virtualized interface for output devices (see next section for details).
- In the split device driver model, the connection between frontend and backend drivers can be broken and thus gives us a point in the data flow path where capability adaptors could be inserted. This requires a small modification in the connection part of original Xen's driver code.
- Xen and Xeno-Linux are open source products that makes it possible to change the system internals for the adaptation mechanism.

3.2 Framebuffer device and xen based virtual framebuffer

Framebuffer device drivers in Linux provide common interface for applications to draw pixels on physical devices without having detailed knowledge of the devices. Typically, framebuffers contain the color values of every pixel on the screen and are mapped into main memory so that they could be read and written through regular memory operations. Internally, framebuffer device drivers use low level system calls (such as inb and outb) or assembly language to operate on the physical devices.

In Xen architecture, virtual framebuffer device drivers are provided to guest operating systems. As is true for other types of device drivers in Xen, framebuffer drivers are also split into two parts: frontend and backend. Frontend part is located in guest domains that serve the guest operating system (as regular framebuffer device in legacy operating systems). Backend drivers are located in the host domain, which connect to frontend drivers when

guest domains start. Backend device drivers map the guest domain's framebuffer requests to physical device driver framebuffer to execute the actual operations (such as reading or writing).

3.3 Other related mechanisms

Linux notifier chain call back mechanism:The notifier chain facility is a general mechanism provided by the Linux kernel. Notifier chains send status change messages to code regions that request them. Unlike hard-coded mechanisms, notifiers offer a versatile technique for dynamically loadable code to get notified when events of interest occur.This mechanism could be utilized to do the adaptations inside the guest OS(though in this work we have not attempted guest OS adaptation).

UPNP protocols:UPNP[1] is a set of device control protocols built upon internet based communication standards, inorder to allow devices to connect and communicate seamlessly among themselves. The protocols describe mechanisms for description, discovery and invocation of services offered by various devices. Also supported are mechanisms for notification of events occuring in various devices. We use open source upnp SDKs for achieving service virtualization and service state migration among compatible applications.

4 System Architecture

Inherent to any service there are three components: an application for providing the service, the necessary hardware resources and an OS which acts as the intermediary. Hence service migration from source machine to destination machine might involve the following scenarios:

- Destination has a compatible(but possibly different) application capable of providing the service: In this case, simple service level virtualiztion comprising of mechanisms for transfering the effective service state to the destination is sufficient.

- Destination does not have a compatible application for providing the intended service: Virtual machine migration functionality available in modern hypervisors solves the issue. But to deal with heterogeneity in the hardware, software needs to adapt to the available hardware.

Using the appropriate migration granularity (service state vs entire VM) for a particular situation in hand is one of the goals of our system and we have designed and implemented a system (represented in Figure 2) for that purpose. In the following discussion we describe the various components of our system

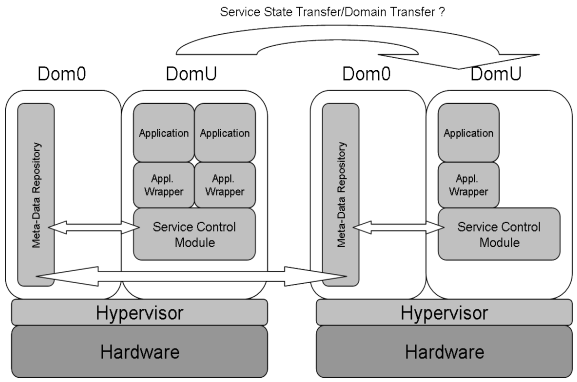


Figure 2: Architecture Overview

4.1 Metadata Repository

The primary functionalities of this are:

- Act as a registry for information about various services

- Provide a mechanism for choosing the correct form of migration required for the situation.

In our architecture, various applications run in different domains and there is a daemon running in the Dom-0 of the machine which acts as a registry for information about these applications(and the services they are intended for). This daemon is termed as the *Meta Daemon* and listens at a designated port.

Inside each guest domain there is a service control module (described in detail in the next section) which gathers information about the services offered by various applications in that domain and registers them with the meta data repository. The service control module sends information about available services in a periodic manner. If the repository does not get any message from a service control module (whose services are currently registered) for a certain amount of time, then it removes the details of the services from that domain.

In order to be compliant to our architecture, an application needs a wrapper which acts like a handle for controlling the application. Service control module interacts with the application through invoking wrapper commands. The application wrapper provides the description of the service offered by the application to the service control module which in turn registers them with the meta repository. Service description comprises of service name, service type, application providing the service, location of the application, the name of the protocol (like UPNP etc) associated with the service etc. Services of the same type and protocol are interoperable. Earlier researches [19, 15] have shown that services following different service protocols could also be made interop-

erable and so our system could be extended to achieve that.

When the user decides to migrate to a new location, it contacts the Dom-0 daemon providing it destination address and the details of the service it wants to get migrated (to the new location). The daemon in the source machine contacts the daemon in the destination machine and queries about the availability of a compatible application. In case of a positive reply, the daemon contacts the service control module in the source machine and informs it to migrate the service to a domain in the destination machine (details explained in the next section). In case of a negative reply virtual machine migration to the destination (and the necessary adaptations) is used.

4.2 Service Virtualization

The two primary components of this layer are (1)Application Wrapper and (2) Service Control Module. Application Wrapper is a small, per-application module that makes legacy applications 'controllable' by our framework. It can either use available API of a particular application or simulate the behavior of keyboard and mouse to send commands to applications just as if a real person is manipulating the application.

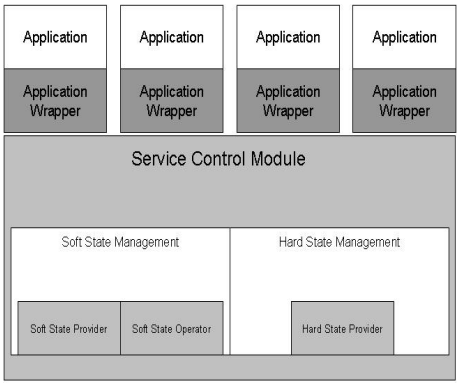


Figure 3: Service Level Virtualization

There is one Service Control Module per domain. Its main functions are (1) collecting necessary information to launch a service (2) pausing/resuming services according to user's need and (3) collecting necessary information for later restoration of the service when user moves. Service Control module contains Soft State Management module and Hard State Management module to help with state storage and retrieval.Soft state refers to the volatile state of the service currently being accessed by the user. An example of soft state is the pause point of a movie when user stops a movie at one location and wants to resume it at another location. Hard state refers to data or preferences stored in persistent storage (such as files). Soft State Provider (in Soft State Management) deals with the storage and retrieval of soft states and Soft State Operator deals with manipulating these states to achieve functions like pausing and resuming of services. In contrast hard state management is primarily concerned with locating the data files associated with a service(for example the movie file in a movie playing service) and how to effecttively retrieve data from them (either by streaming data from them if possible or by downloading the data file.)

If at the time of migration the meta data repository decides about using this mode of adaptation mechanism for service migration, it sends out a PAUSE request to the Service Control module. Service Control pauses the service, retrieves the soft state and transfers the soft state to the Service Control Module in the destination. When the user is ready to resume the (paused)service in the destination, it needs to to provide a RESUME request to the service control module in the destination. Service Control then interpretes the soft state and instructs the corresponding application wrapper to resume the execution of the service.

4.3 Adaptation by System Virtualization

In case the meta data daemon is not able to locate a compatible application in the destination, domain migration is used. As has been explained in earlier sections some form of adaptation might be necessary for proper functioning of the applications (inside the migrating domain) in the destination machine. In one form, the adaptation might happen inside Dom0 (which has direct control over the hardware resources). Otherwise, if the application has the ability to adapt, the adaptation might happen inside the migrating domain.

4.3.1 Back end adaptation

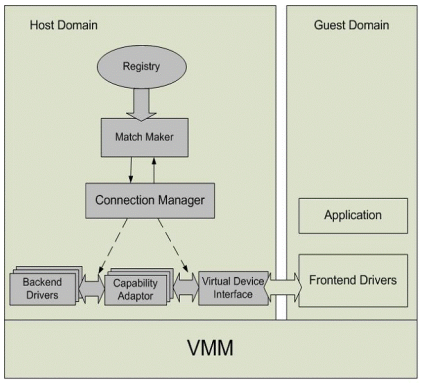


Figure 4: Back End Adaptation

As shown in the figure, we have a pool of backend device drivers that control physical devices. Different than original Xen backend drivers, these drivers in our system do not connect to frontend drivers directly. Instead, the connection manager connects them to appropriate capability adaptors (discussed later) and then to an instance of the virtual device interface, which frontend drivers can discover and connect to when the guest OS comes online. After the connection is made, data flows without the involvement of the connection manager.

We also have a registry that keeps information about all the capability adaptors that has been registered in our system. The match maker takes the information and decides which algorithm to use at run-time based on the information provided by the connection manager. For example, the connection manager can tell the match maker that the virtual device interface expects an 800*600 screen but the only backend driver available is for a 400*300 display. The match maker may take the shrinking algorithm in this case. The connection manager can take the decision made by the match maker and connect the appropriate adaptor with other system modules.

Capability adaptation is done by breaking the frontend/ backend connection and adding capability adaptors (CA) between them. In this case, FE issues requests to CA instead of directly to backend drivers. CA can do adaptations on the requests based on its information about the frontend and backend and then forward the requests to backend to get processed. For example, if the guest OS was originally connected to an 800*600 display but is now migrated to a 400*300 display, the capability adaptor, by knowing that information, can shrink the frame buffer accordingly when frontend issues the updates on the 800*600 frame buffer (frontend still thinks it has an 800*600 screen in this case). Similarly, another capability adaptor can expand the screen if the destination framebuffer is larger than the original.

Our system design meets all the design principles that we presented in section 2. All the modules in our system are in the host domain and we keep the guest domain unchanged and generic for the migration. The registry in our system allows the dynamic registration of capability adaptation algorithms, which can be used later for selection. Removing an entry in the registry prevents future selection of the corresponding algorithm and therefore is an uninstallation of the adaptor. The match maker in our system takes charge of the algorithm selection for particular adaptation needs. The connection manager can dump the device states by calling the the virtual device interface and pack the state with the domain image for later resumption of the domain on the destination.

Device Discovery: In Xen it means the actions taken by the frontend drivers to find the corresponding backend driver when the guest domain comes online. In original Xen system, Frontend drivers do it by writing to the appropriate device type's entry in Xenstore that backend drivers have watches on. After that, FE and BE communicate the basic connection information through Xenstore to set up event channels and shared pages to make the connection. In our system, we separate the discovery from the connection establishment. The connection manager has watches on the entries in Xenstore and waits for frontend drivers to discover. After it sees a frontend driver coming, the connection manager initiates an instance of virtual device interface and hands it over to the virtual device interface for the connection establishment with the frontend. The separation of discovery from connection establishment moves the connection manager out of the common path for data transfer while still setting up a central point of contact for discovery.

The virtual device interface takes charge of the connection with the frontend driver as described above. It sets up the event channels and shared pages with the frontend for later data communication based on Xen's mechanism. The virtual device interface also takes suggestions from the connection manager on which capability adaptors it should connect to and which backend driver will handle the requests. In our current implementation, we took the connection code from original Xen's backend (which is used for connecting to frontend drivers). We use dynamic linked library for the capability adaptors in order to be able to load them at run-time. The CA library names are recorded in the registry and will be provided to the virtual device interface upon request. These CA libraries are required to implement several functions in order to make the connection establishment happen successfully.

Virtual to Physical Device Mapping:Virtual device interface is mapped to physical device backend dynamically at run-time. At the first time the guest domain starts, the connection manager chooses the best resources for the guest domain and therefore maps the virtual device interface to the best possible physical resource backend. If the guest domain is migrated from somewhere else, the virtual device interface learns the source device specification from the frontend drivers and passes it to the connection manager. The connection manager then selects the physical resource backend that best matches that specification (e.g., screen size, resolution) for the guest domain. If a match is found, then the mapping is made directly from the virtual device interface to the physical resource. However, in most cases, the exact match cannot be found. Therefore, we need capability adaptation to solve the heterogeneity problem.

Capability Adaptation:In case the virtual device is different from the physical device, capability adaptation has to be done by capability adaptors. Capability adap-

tors are classified by categories based on their functionalities. For example, video adaptors are able to adjust frames based on their original size and target size. Keyboard adaptor can change the key codes accordingly to make the source and destination input device look alike to users. Each adaptor has to implement a certain set of functions that are required for its category. This set of unique functions implemented by all adaptors in a category enables the dynamic change of adaptors in this category at run-time. The implementation details of these functions are presented in the implementation section later. Note that the adaptation algorithms themselves are not the focus of our system and therefore are not the main contribution of this work. Our goal is to enable the dynamic installation and selection of capability adaptor algorithms at run-time to ensure the seamless migration of user activities.

Data Flow:

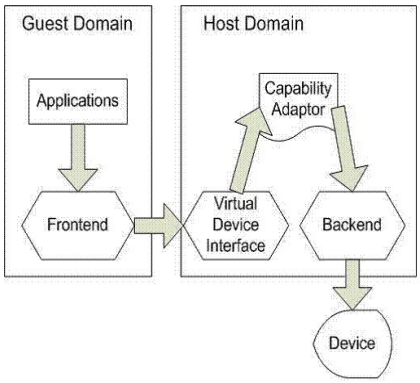


Figure 5: Data Flow

Different types of devices have different data flow in Chameleon. Figure 5 shows the typical data flow of an output device. Applications in guest domain issue requests for writing to a particular area on the screen. The requests are sent to the frontend virtual framebuffer device in the guest domain, which connects to the virtual device interface in host domain. Typically, a capability adaptor is needed and the screen update requests are forward to the capability adaptor to do certain adaptation before they reach the physical backend device for the actual updates. Note that the connection manager is not in this common path. Therefore, we do not have additional management overhead in the entire process. Compared to data flow in the original Xen virtual framebuffer system, the only additional module that the requests need to go through is the capability adaptor. However, this module is not even required if the virtual device interface matches the physical device backend exactly.

Device State Migration:There are a variety of internal states of backend drivers that may be necessary to be migrated in order to ensure the seamless migration.Framebuffer contents, in display device drivers is one example of such states. These states are internal to physical device drivers and out of the control of guest domains. Therefore, only migrating the guest domain may cause these states to be lost. We need to capture and resume such states with the migration of guest domain in order to provide the real unstopped service to mobile users. Device state is captured by the connection manager when the migration is triggered. The connection manager collects all the internal states of the physical backend driver and packs them up with the migration of the guest domain. At the destination, the connection manager extracts the states and dumps them into the selected physical backend accordingly. These states may also need to be adapted if the source and destination devices are not fully compatible.

4.3.2 Front-end Adaptation

In the case of domain migration, the necessary adaptation could also happen inside the migrating domain (front end). If we are concerned with adapting to varying display device configurations, the front end contains applications like windowing system, media renderer etc., some of which have inbuilt adaptation mechanism that enables them to adapt to varying configurations. For example, the Xorg windowing library contains both device independent (DIX) and device dependent (DDX) interfaces. The DDX is responsible to handle different kinds of framebuffer hardware and the DIX contains the logic/algortihms for scaling, drawing and other windowing effects. In cases like this when the application itself is capable of adapting there is not much need for backend based capability adaptation. But owing to the virtualized nature of our architecture, the application does not directly interact with the underlying hardware. So there is a need for the collection of information about the hardware and relay it to the application.

A brief description of the various steps involved in adaptation at front-end is as follows:a)Migration of frontend to new VMM.b)Backend declares the available resource parameters(screen resolution).c)Frontend reads the parameters from xenstore.d)Frontend reconfigures framebuffer memory area. e)Fb reconfiguration is informed to upper modules by the kernel callback notifiers.Similarly the related user space processes (mmapers) are sent a SIGWINCH signal informing about the change.f)Finally frontend connects to backend by making a new event channel.

4.3.3 Why front-end adaptation is not the right approach

Application based adaptation inside front-end is highly dependent on the ability of the applications to adapt to varying environments. Not all applications are written with this in mind. So any migrating domain might have some applications which might know how to adapt and some which lack the inbuilt mechanism to adapt to varying IO device characteristics. Also there is no neat way to determine which particular applications have the adapting capability. As a consequence, trying to push the responsibility of adaptation into the front end does not work in a generic environment.So even though in this particular case of adapting to varying display configurations, front-end based adaptation has performance benefits (as explained in section 6.2.1), we decided to proceed with back-end based adaptation.

5 Implementation

5.1 Service Level Virtualization

Application wrapper and service control module are the two primary components in achieving service level virtualization. In our system we have used UPNP style service discovery and invocation. Many applications available already are UPNP enabled. We also created an UPNP compliant wrapper for mplayer and tested our system.

Other protocol based service discovery could also be integrated into our system by making the various service control points aware of them and providing interoperability between the various protocols. But for our proof of concept implementation UPNP had some advantages like readily available SDKs, sample control point implementations etc.

Keeping in synch with the UPNP implementation, XML representations are used for service description and discovery. The softstate is also represented as a XML file. This facilitates softstate storing and transfer either online or by physically carrying and feeding it in a new environment.

5.2 Virtual Device, Physical Resource and their mapping

We use different data structures to represent virtual device, physical backend device (resource) and the mapping between them. Below are these data structures.

```
struct virtdevice_
{
   int devType;
   char devName[64];
   int devID;
   int domID; //DomU id
   struct xenfb* xenfb; //virtual FB
   void* extraInfo;
   struct virtdevice_ * nextdev;
};
```

The virtual device structure records the information of the virtual device interface. Virtual device is discoverable by the frontend drivers from its device type (devType) and device name (devName). The xenfb field in the structure is used for virtual framebuffer to identify the framebuffer it uses. The virtual device structure is a linked list and the nextdev pointer points to the next node in the linked list.

```
struct resource__
{
  const char* resName;
  int resID;
  int resType;
  int resStatus;
  void* extraInfo;
  adapter_t* padapter;
  void (*useResource)(void*);
  void (*reuseResource)(void*);
  struct resource__* next;
};
```

The physical driver (resource) structure is an internal structure of our system. We use it to maintain the physical backend device driver's information and whether it has been used or not (resStatus). The two function pointers in the structure point to two required functions of the physical backend device driver that could be used to utilize the device and resume the device states. The extraInfo field stores device specific information for each device category. The resource structure is also a linked list and the next pointer points to the next node in the list.

```
struct mapping__
{
  int mapID;
  int backendID;
  int frontID;
  state_t MapState;
  pthread_t tid;
  pthread_mutex_t mapMutex;
  virtdevice_t* pVirtdev;
  resource_t* pResource;
  adapter_t* pAdapter;
```

```
  struct mapping__* pnext;
};
```

The mapping structure stores the mapping information between physical backend resources, capability adaptor and virtual devices. Since our system creates a separate thread to handle the requests for a particular mapping in order to avoid the interference between devices, this structure also records the thread information as well as a mutex for the thread. The mapping structure is a linked list and the pnext pointer points to the next node in the list.

5.3 Capability adaptor interface

In our system, we consider one type of capability adaptation: frame buffer adaptation for display device. The adaptors have to implement certain functions to make the link to these libraries happen at run-time. Below are the code for these functions and the frame buffer update structure definition:

```
updateDisplay(void* src_buf, void*
dst_buf,struct fbu *src, struct fbu
*dst);

updateKB(char *src_code, char *dst_code,
int src, int dst );

struct fbu
{
  //Update Area info
  int x;//start point in x axis
  int y;//start point in y axis
  int w;//width of the update area
  int h;//height of the update area

  //Setting info
  int size_w;//width of the screen
  int size_h;//height of the screen
  int r;//color setting for red
  int g;//color setting for green
  int b;//color setting for blue
};
```

These two functions are self-descriptive: they take the source frame buffer and convert it into destination based on the source and destination information (src and dst). The updateDisplay function also sets x, y, w and h field in the dst as an output since these values will be used later to update the actual device. (For example, if the source device is 800*600 and it updates the area ((100,200),(240,300)), then the destination device with a screen size of 400*300 should update the area of ((50,100),(120,150)) accordingly.)

5.4 Registry and Match Maker

We store the capability adaptor information into our registry for the match maker to select at run-time. Each entry in the registry contains two parts: (1) capability adaptor library name and (2)a flag field that shows the types of adaptation that could be performed by the corresponding algorithm. The flag has 5 binary bits that represent 5 types of adaptations:

- Extend the screen horizontally
- Shrink the screen horizontally
- Extend the screen vertically
- Shrink the screen vertically
- Do color setting change

For example, an adaptation algorithm that is very good at extending the screen both horizontally and vertically but cannot do the color setting change will have a flag of 10100 in the registry. Another algorithm that is particularly good in shrinking the screen vertically and can do color setting change will have a flag of 00011.

At run-time, the match maker takes the source and destination device information and selects the appropriate algorithm based on the flags in the registry. It passes the library name to the connection manager for loading and connecting to other modules.

We also provide a small tool for adding/removing entries to/from our registry for the capability adaptors. Basically, the developers of the adaptation algorithm compiles the algorithm code to get a dynamically linkable library. Then they can use our tools to specify the types of adaptation of the algorithm and the library name to add it into our system. Our tool can also allow users to delete an entry from the registry in case the adaptor library is removed or no long used.

5.5 Frame Buffer

As we discussed before, we choose to use virtual framebuffer support of Xen in Chameleon. However, virtual framebuffer device drivers are always mapped to physical framebuffer in order to draw pixels on physical devices. In the current implementation of Chameleon, we use two types of framebuffer devices: a simulated framebuffer device using VNC server, and a physical framebuffer device using NVIDIA framebuffer device drivers.

The simulated VNC-enabled framebuffer device driver starts a VNC server and uses it as the target device for display. This framebuffer device can be initialized and updated as regular physical framebuffer devices. Different from physical framebuffer device, this simulated device draws pixels in the VNC server framebuffer

instead of drawing in physical framebuffer on physical devices. Any VNC client can connect to the VNC server to see the screen that is supposed to display on the physical device. The advantages of using the simulated driver are the following:

- Original Xen's virtual frame buffer driver is also VNC server based. So using a VNC based frambeuffer in our experiment helped us to use existing Xen codes.
- The simulated VNC framebuffer and driver enables us to run and test our code in the user level thus making development testing and debugging much easier.
- Functionally the simulated framebuffer driver is equivalent to a physical framebuffer driver. The only difference is that instead of sending output to a physical device, it sends to VNC server.

In addition to simulated framebuffer, we also have integrated a NVIDIA framebuffer, a physical device driver into our system. The primary goal is to test the effectivenes of our approach in the real scenario. The physical framebuffer, similar to the simulated framebuffer, could be initialized and updated to reflect any changes on the screen. Internally though its a bit different from the simulated framebuffer and uses lower level system calls and assembly code to operate the video card on the machine. A brief performance comparison beween using a simulated driver and physical drivers is presented in the evaluation section.

6 Performance

In order to measure the effectivenes of our system, we have conducted experiments in varying scenarios. We use two identical AMD Athlon (TM) 64X2 dual-core machines with 1G memory and with Ethernet connection to each other. Both machines have Fedora 8 Linux with Xen 3.1 installed. They contain other necessary libraries necessary for our experiments.

6.1 Service Level Virtualization Performance

Although service level virtualization is an integral part of our overall system architecture, quantitatively measuring its effectivenes is not easy. In our approach of XML based representation of services and intermediate service states, the size of the XML file which needs to be transferred across domains as the carrier of softstate and the time taken during the process is one representative evaluation. In our sample movie playing service, the file representing the intermediate softstate was 3.6KB and the entire process of pausing, file transfer and service resumption took 946msec. While the actual size of the XML file will differ from one type of service to another and also on additional information added to it for hard-state management, our guess is that the size will always be of the order of kilobytes and the latency involved will never cross 1.5 sec. We assume that a 2 second latency between initiating an action and observing its effect is tolerable from a user experience point of view.

The real advantage of a service level state migration is that it prevents us from migrating an entire VM image and thus reduces both amount network data transfer involved and the latency involved in the resumption of service.

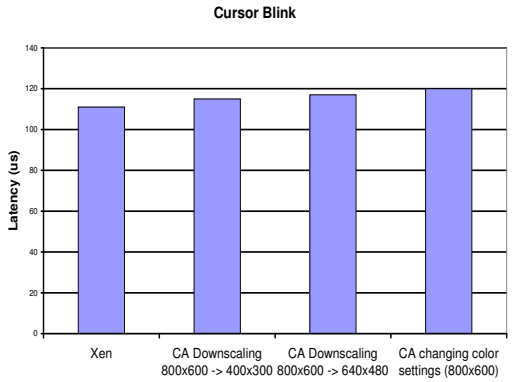
6.2 Performance of Capability Adaptation

The first experiment is to evaluate the overall performance of our system with capability adaptation compared to original Xen's split device driver performance. We measure two types of costs in this experiment: initial set-up cost and screen update cost. The initial setup cost includes the cost for connection establishment and frame buffer initialization. The screen update cost is the cost to update a certain area of the frame buffer after its initialization. In the current system, the entire frame buffer is NOT updated every time there is a change on the screen. Instead, only the portion of the frame buffer that is changed (usually only a small area) is updated in the framebuffer devices. We called this type of cost as "cursor blink".Figure 6 shows our results for different capability adaptors.

In the figure, we can see that for a cursor blink, our system with different capability adaptors has similar performance as original Xen's virtual frame buffer device. Therefore, for the common frame buffer updates, we don't introduce a lot of overhead by adding the capability adaptation feature in the system.

However, the initial setup of Chameleon introduces noticeable costs compared to original Xen. The reason for such overhead is mainly due to the memory copy cost: the initialization of the capability adaptor needs an additional memory copy to tune the display before the initialization. Figure 7 shows the details procedure and Figure 8 shows the time for copying different sizes of the memory.

We think such a memory copy is inevitable for capability adaptation since the "internal copy" shown in the figure is done by the device specific code (in our case, the VNC server library for the simulated framebuffer and physical device driver code for physical framebuffer). Unless we fully understand the internal mechanisms of the device and control codes, we won't able to change



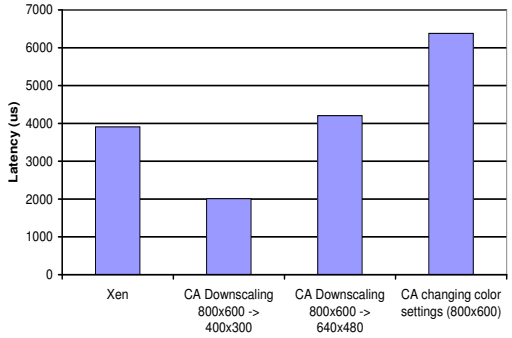


Figure 6: Performance of Chameleon

this device specific memory copy and therefore cannot avoid such a copy to the internal device frame buffer (simulated or physical). However, we notice that such a memory copy is only a one-time cost for the initialization process and will not commonly occur in the frequent screen updates. We also plan to investigate the device drivers internals to get rid of this memory copy by changing the driver internals.

6.2.1 Back-End Adaptation vs Front-End Adaptation

As has been mentioned in section 4.3.2, the adaptation could happen either inside the back-end or could be handled by the applications themselves inside the front-end. We performed an experiment to measure the relative cost of display adaptation among the two approaches, the results of which are presented in Figure 9.

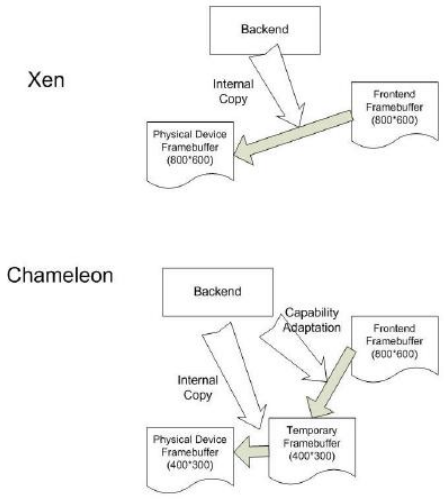


Figure 7: Additional Memory Copy for Chameleon

The primary reason for higher latency in case of back-end adaptation is the extra memory copy involved. Front-end adaptation does not involve this extra memory copy and hence is faster. But despite this advantage, for reasons described in section 4.3.3, this is not a practical approach.

6.3 VNC framebuffer vs physical framebuffer

The second experiment is to measure the performance difference between the physical framebuffer and VNC framebuffer. In this experiment, our physical framebuffer has the size of 640*480. Therefore, we compared the overhead of VNC framebuffer with the same size and conducted evaluation on both initial setup cost and cursor blink cost. Figure 10 shows our results.

In the figure, we can see that both cursor blink and initial setup cost of the physical framebuffer are less than the VNC framebuffer. We think that difference is due to the relatively slow in-memory processing of VNC framebuffer since it needs to simulate the behavior of physical framebuffer. The physical framebuffer, in contrast, doesn't need such a simulation and therefore performs faster than VNC framebuffer.

We also note that although there are differences between VNC framebuffer and physical framebuffer, the overheads of both framebuffers are relatively small compared to original Xen's framebuffer without capability adaptation (it even performs better than original Xen for physical framebuffer since Xen originally uses VNC framebuffer). Therefore, no matter which framebuffer is used, the performance of Chameleon is comparable

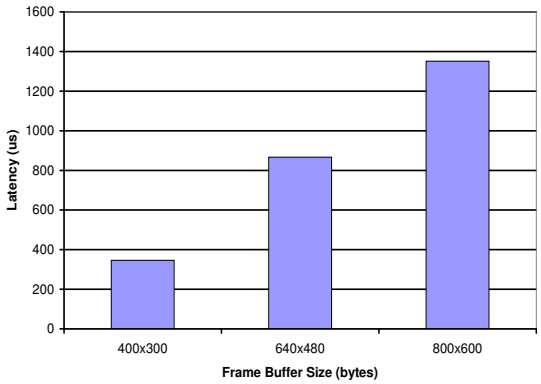


Figure 8: Memory Copy Cost

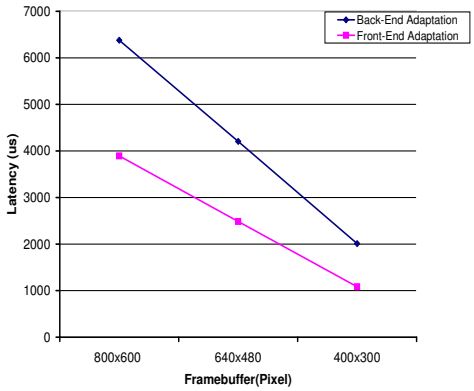


Figure 9: Front-End vs Back-End Adaptation

to Xen virtualization system with the added capability adaptation functionality.

6.4 State Migration

The last experiment is to measure the cost for migrating the frame buffer contents from one device to the other. In this experiment, we started two simulated VNC-enabled backend driver and make one of them connected to the frontend driver and display the screen. Then, we issued a command to the running backend driver to make it migrate to the other VNC-enabled driver at run-time. The results presented in Figure 11 show the costs for such a migration (excluding the initial setup cost for both drivers, which is shown in the previous experiment).

In the figure, we can see that most of the cost for such a migration is due to the memory copy cost of the frame buffer from one driver to the other. The additional overhead is minimal in these scenarios. Since such a memory copy is inevitable for the device migration and our system only added a small amount of overhead, we think Chameleon deals with these migrations in an efficient way. We would also like to point that these migrations are not common system operations and an overhead of few milliseconds is always acceptable.

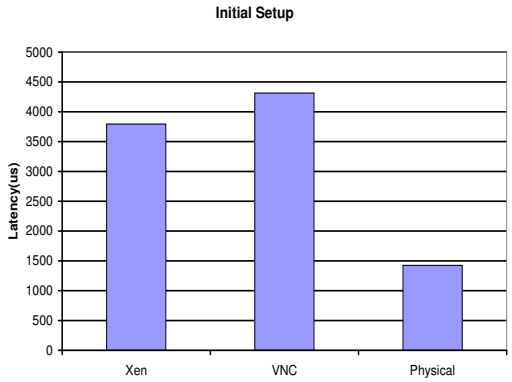
7 Related Works

Ours is definitely not the first attempt at achieving seam less mobility. Enabling users to move seamlessly from one environment (/device) to another has been an active research topic for quite some time. People have tried to approach this issue by making modifications at various layers of the software stack (application layer/middleware/within OS).

The AURA task migration system described in [20] is one of the earliest attempts at defining task abstraction at the application layer. It is conceptually similar to the service level virtualization where the same service could be provided by different application in different environments.

In [17], Satyanarayanan et al. have identified the basic components of an infrastructure essential for providing seamless mobility. They have implemented a system for achieving seamless mobility using a technique similar to VM migration and have devised a mechanism for fine grained transfer of virtual machine disk contents to a new location. In another related paper [23] the authors have tried to use mobile devices to transfer small residual VM images from one environment to another. The environment provides base VM for performing a particular task. The approach is similar to our service virtualization model but is applied to VM migration to minimize the amount of data transfer.

In [8], Edmonds et al. discuss about various models for the structuring of an adaptive system. An application-transparent model performs all the adaptation at operating system level. An application-specific adaptation model places all the responsibilities with the application. The first approach has the advantage of centralized resource control but treats the application as a black box. The second approach might lead to contention for resources among applications due to lack of centralized coordination. An integrated approach is also possible where the resource monitoring(and allocation) is done by operating system and adaptation is performed by the application.The authors have developed a framework for



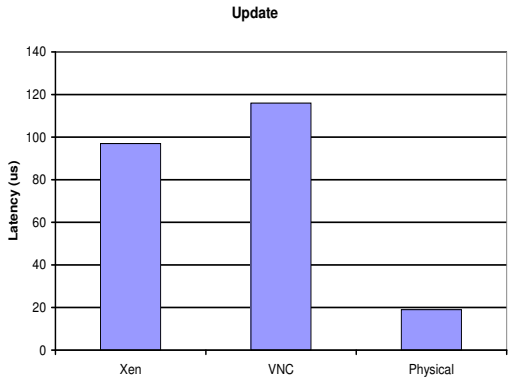


Figure 10: VNC framebuffer vs Physical framebuffer

development of application adhering to this integrated approach. The work by Becker et al. [4] mentions about the requirements for adaptation at various levels. The devices available to an application and also the characteristics of a device may vary over time and location. An application needs to adapt to these changing scenarios. Device capabilities as well as characteristics of local and remote services need to be uniformly accessible to the application.

Proper representation of the hardware and software context in which an application is operating is very essential for adaptation to take place. W3C's Composite Capability/Preference Profiles (CC/PP) is predominantly used for this purpose. Buchholz et al. [5] describe an alternate language named Comprehensive Structured Context Profiles for context representation. The primary objective of this work is to provide highly structured repre-

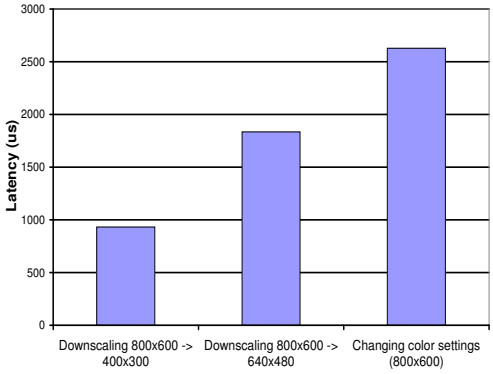


Figure 11: Device State Migration Costs

sentation of context information for easing the task of dynamic composition(/decomposition) of context profiles. In the DCOD framework [10], Fu et al. have described a model for enabling an application to make use of various devices as and when they become available. The framework virtualizes the access to devices and has a matchmaking engine which matches the application requirements with the capabilities of devices and selects a device (or a group of devices) for the purpose.

Dynamic Composable Computing [22] targets at the adaptation in a different perspective. Users of DCC can choose the environmental resources to connect to their mobile devices and take advantages of these resources to enhance their experience. The adaptation is done by user's manually choosing the appropriate resources to match the application requirements. Different from DCC, our system uses an automatic mechanism to choose capability adaptation algorithms for users based on the application requirements and the environment resource information. While DCC focuses on the usage of surrounding resources for mobile users, Chameleon targets at the adaptation mechanisms for service migration across heterogeneous platforms. The two projects are orthogonal and could be complementary to each other.

While not representing a truly seamless mobility solution, Microsoft Remote Desktop, Virtual Network Computing (VNC) [16] etc are thin client approaches to allow a user's display to be dynamically moved from one machine to another. Xmove [9] enables users to map the virtual X server to different physical X servers so that the IO state could be migrated. These solutions are appropriate for their goals and are at best complementary to our goal of seamless mobility. Also starting with version 3.2, Xen provides a mechanism to dynamically change the virtual

framebuffer size. In this case the change request needs to originate from the application in guest domains and is different from our goal of front-end agnostic adaptation.

8 Conclusion and Future Works

In this paper, we have presented an integrated approach towards achieving seamless mobility. Using movie playing as a sample application we have designed and implemented a system for achieving seamless mobility . Content migration is a must for achieving seamless mobility but making the migration at the right granularity is important and has been one of the goals of this effort. Virtualizing the service at application layer and thus achieving mobility by migrating service state is definitely light weight and preferable. But this approach is not feasible at all times due to unavailability of suitable applications in the destination environment. Assuming the availability of a virtualized environment, virtual machine migration could be used in such cases. But with VM migration comes the added burden of making it work in the new environment where the characteristics of IO devices might be different from the original environment. We have introduced a concept called capability adaptation which is realized in our system by dynamically inserting capability adaptor modules in the IO data path. Through performance evaluation, we demonstrate that our system introduces minimal overhead to the existing Xeno-Linux setup.

There are quite a few related avenues in which we plan to conduct further research. In the service level virtualization we have currently looked at only UPNP service discovery protocol. We will work on integrating other protocol families like AVAHI etc into our system. We will also try to port our system onto mobile platforms such as PDAs in order to support migration onto those platforms. Cross architecture migration (for example between x86 based PCs to ARM based PDAs) is currently not possible and is definitely worthy of further research.

References

[1] Universal plug and play device architecture.

[2] BALAN, R., FLINN, J., SATYANARAYANAN, M., SINNAMOHIDEEN, S., AND YANG, H.-I. The case for cyber foraging. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2002), ACM, pp. 87–92.

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T. L., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP* (2003), pp. 164–177.

[4] BECKER, C., AND SCHIELE, G. Middleware and application adaptation requirements and their support in pervasive computing. In *ICDCS Workshops* (2003), pp. 98–103.

[5] BUCHHOLZ, S., HAMANN, T., AND H?, G. Comprehensive structured context profiles (cscp): Design and experiences. *Pervasive Computing and Communications Workshops, IEEE International Conference on 0* (2004), 43.

[6] CÁCERES, R., CARTER, C., NARAYANASWAMI, C., AND RAGHUNATH, M. Reincarnating pcs with portable soulpads. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2005), ACM, pp. 65–78.

[7] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 273–286.

[8] EDMONDS, T., HOPPER, A., AND HODGES, S. Pervasive adaptation for mobile computing. In *ICOIN* (2001), pp. 111–118.

[9] ET AL., E. S. Xmove: A pseudoserver for x window movement, 1994.

[10] FU, R. Y., SU, H., FLETCHER, J. C., LI, W., LIU, X. X., ZHAO, S. W., AND CHI, C. Y. A framework for device capability on demand and virtual device user experience. *IBM J. Res. Dev. 48*, 5/6 (2004), 635–648.

[11] GRIMM, R., DAVIS, J., LEMAR, E., MACBETH, A., SWANSON, S., ANDERSON, T., BERSHAD, B., BORRIELLO, G., GRIBBLE, S., AND WETHERALL, D. System support for pervasive applications. *ACM Trans. Comput. Syst. 22*, 4 (2004), 421–486.

[12] GU, X., MESSER, A., GREENBERG, I., MILOJICIC, D., AND NAHRSTEDT, K. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing 3*, 3 (2004), 66–73.

[13] KIENTZLE, T. Scaling bitmaps with bresenham, 1995.

[14] M. SATYANARAYANAN, E. A. Pervasive computing: Vision and challenges. *Pervasive Computing and Communications Workshops, IEEE International Conference on* (August 2001), 10–17.

[15] NAKAZAWA, J., TOKUDA, H., EDWARDS, W. K., AND RAMACHANDRAN, U. A bridging framework for universal interoperability in pervasive systems. In *ICDCS* (2006), p. 3.

[16] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., WOOD, K. R., AND HOPPER, A. Virtual network computing, 1998.

[17] SATYANARAYANAN, M., KOZUCH, M., HELFRICH, C., AND O'HALLARON, D. R. Towards seamless mobility on pervasive hardware. *Pervasive and Mobile Computing 1*, 2 (2005), 157–189.

[18] SMITH, A. R. A pixel is not a little squirrel, 1995.

[19] SONG, X., AND RAMACHANDRAN, U. Mobigo: A middleware for seamless mobility. In *RTCSA* (2007), pp. 249–256.

[20] SOUSA, J. P., AND GARLAN, D. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA* (2002), pp. 29–43.

[21] SUH, S.-B., SONG, X., KUMAR, J., MOHAPATRA, D., RAMACHANDRAN, U., YOO, J.-H., AND PARK, I. Chameleon: a capability adaptation system for interface virtualization. In *MobiVirt* (2008), pp. 18–22.

[22] WANT, R., PERING, T., SUD, S., AND ROSARIO, B. Dynamic composable computing. In *HotMobile* (2008), pp. 17–21.

[23] WOLBACH, A., HARKES, J., CHELLAPPA, S., AND SATYANARAYANAN, M. Transient customization of mobile computing infrastructure. In *MobiVirt* (2008), pp. 37–41.