Chameleon: A Capability Adaptation System for Interface Virtualization

Sang-bum Suh [1], Xiang Song [2], Jatin Kumar [2], Dushmanta Mohapatra [2],
Umakishore Ramachandran [2], Jung-Hyun Yoo [1], IlPyung Park [1]

[1] SW laboratories, Samsung Electronics, Korea
[2] Georgia Institute of Technology, Atlanta, GA, USA

sbuk.suh@samsung.com, songx@cc.gatech.edu, jatin.kumar@gatech.edu, dmpatra@cc.gatech.edu
rama@cc.gatech.edu, yjhyun.yoo@samsung.com, ilpyung.park@samsung.com

Abstract

As computing capabilities of embedded devices increase, services such as Internet and multimedia are comportably supported by such devices. Users may demand for migrating services from one machine to another that can provide better capabilities and convenience in terms of battery stock and input/output (I/O) method. However, I/O devices capabilities between embedded mobile devices and user interaction behavior are particulary different. So, heterogenous I/O capabilities and functions may make migration of computing states between embedded devices virtually impossible. To guarantee seamless I/O state migration between embedded mobile devices, we propose interface virtualization and capability adaptation. In this paper, we describe a system architecture based on Xen virtualization as well as design principles and performance.

1 Introduction

People nowadays tend to use their mobile devices to enable their work any place at any time. However, mobile devices are usually constraint by its limited power, storage and screen size. People carrying mobile devices may want to take advantage of nearby resources in the environment to enhance their experience. For example, a frequent traveller may want to use a high definition TV equipped in an airport lounge to see his movie that stores in his personal digital assistant (PDA). In a typical ubiquitous computing environment, these I/O devices becomes pervasive in any surroundings, which allows people to connect and use. We need a comprehensive infrastructure that can enable the connection between people's mobile devices and the environment.

In addition to use the environment provided resources, mobile users may also want to seamlessly migrate their work from place to place. Taking the above example, if the person who is watching a movie at the lounge doesn't finish it before boarding, he may want to "move" his movie to the small monitor in front of his seat to continue it without worrying about where he left off. In this case, his PDA will coordinate with different environments (airport lounge and the airplane) to transfer the internal "state" of the movie player from one place to the other in order to migrate such a movie playing service seamlessly.

One important issue for such a migration is that the source and the destination I/O devices are always heterogenous. Output devices may have different sizes/resolutions, color settings and even supports for complex operations such as 3D. 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 adapt different capabilities of the source and destination devices to make the migration as seamless as possible.

In this paper, we will introduce a system, called Chameleon, that can dynamically adapt the capabilities of different I/O devices by allowing the dynamic installation of capability adaptors. We provide transparent adaptation to applications with no additional requirements for the application to be aware of it. However, we are not trying to develop an intriguing capability adaptation algorithm. A lot of work in HCI [6][7] is orthogonal to our work and can be complementary to our system for best user experience. Our work is also in parallel with application level adaptations such as [1][3], which requires applications to be aware of the device change and therefore can better adjust their UI for the device change. These projects are also orthogonal to our work and our system can work better if applications are not designed to allow such adaptation.

The rest of the paper is organized as the following: after presenting our design principles in section 2, we briefly describe the Xen and why we choose Xen as the

base system in section 3. Then, in section 4 we present our design, followed by details of implementation in section 5 and the performance evaluation in section 6. Finally, after briefly summarizing other people's related projects, we conclude our paper and describe possible future work.

2 Design principles

Since we consider the migrations of people's activities across environments, in our system, there is always a mobile platform (guest) that is moved with user and a stationary environment system (host) that can provide resources locally to user. The guest may be migrated to different host systems if user moves, where it may need to migrate all its previous states to a new host and continue its activities by using new resources provided by the new host.

The first design principle is that we want to keep the guest unchanged and make the host adapt to the guest. Since in a typical ubiquitous environment, the resources are always heterogeneous and therefore the adaptation has to happen in order to enable the seamless migration. Instead of making the guest adapt to the host, it is always better for the host to adapt the guest since the host always knows the local resources better and therefore understand how to adapt a generic guest using its local resources.

The second design principle is that we want to build a system that can support the dynamic installing and uninstalling of different capability adaptors for dynamic adaptation. The adaptation algorithms, however, are not our main concern and are out of the scope of this system. We rely on other researchers (probably in HCI) to develop the algorithm for adaptation.

The third design principle is that we want to make the selection of adaptation algorithms automatic and at run-time. We notice that there are usually a variety of adaptation algorithms that can be supported and choosing one algorithm over the other is always a challenge. A successful system should support the selection of an algorithm when the actual requirements come.

The fourth design principle is that we want to make the migration as seamless as possible. Therefore, we may need to migrate the device states (such as frame buffer for display devices) along with the domain migration. By dumping and resuming the device states, we can ensure the seamless migration of the domain without losing any states.

3 Xen and Virtual Device

We choose Xen as our base operating system to implement our capability framework. Xen is an open-source virtualization system that can support multiple guest operating systems (guest domains) running on top of a hypervisor (also called a virtual machine manager) and sharing the hardware resources. Xen always needs a host operating system (usually Xeno-Linux) running on top of the hypervisor to take controls of the guest operating system. By virtualizing the hardware resources, the hypervisor gives all the guest operating system the illusion of having total control of all resources while internally it manages the sharing of the resources across different guests. In addition, the hypervisor creates different domains for these operating systems in separated address spaces, called virtual machines (VM), and Xen provides facilities to suspend and resume these VMs.

Split device driver model is a key feature in Xen that make the capability adaptation happen without significant change of the system. Figure 1 shows the basic architecture of Xen system. We are aware that the Xen split driver model consists of front-end drivers, back-end drivers and physical (or native) device drivers. However, we assume that a backend driver in this paper includes a native device driver in order to simplify the architecture shown in Figure 1. There might be one or more guest operating system with generic frontend device drivers (FE) that seek to access those devices controlled by the backend. Frontend drivers connect to backend drivers when the guest OS is loaded into the system. After a complex handshaking process and connection establishment, FE can forward the application requests to access the device (such as reading or writing) to BE. BE then processes those requests to fulfill FE's needs, just as what traditional device drivers do.

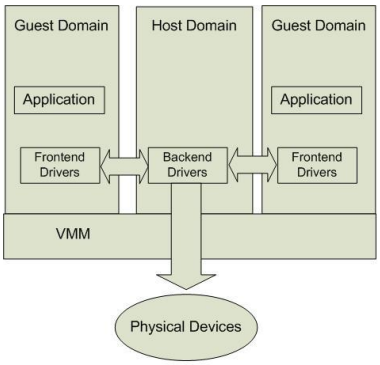

Figure 1: Xen Architecture

In such a split device driver model, the frontend driver can be seen as a *virtual* device driver that is very generic and takes no device specific information. The virtual device driver becomes concrete when it connects to a backend, which has a lot of information about physical device such as resolution and color settings. Applications running in the guest operating system only see virtual device drivers and don't need to worry about

device specific setting and configurations. This gives application developers the freedom to focus on their application logics instead of dealing with physical device related issues.

4 System Architecture

In our system, 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 BE directly. CA can do adaptations on the requests based on its information about the FE and BE and then forward the requests to the BE to process (e.g. changing an 800*600 screen to a 400*300 screen then giving the new screen for BE to display). Figure 2 shows our system architecture.

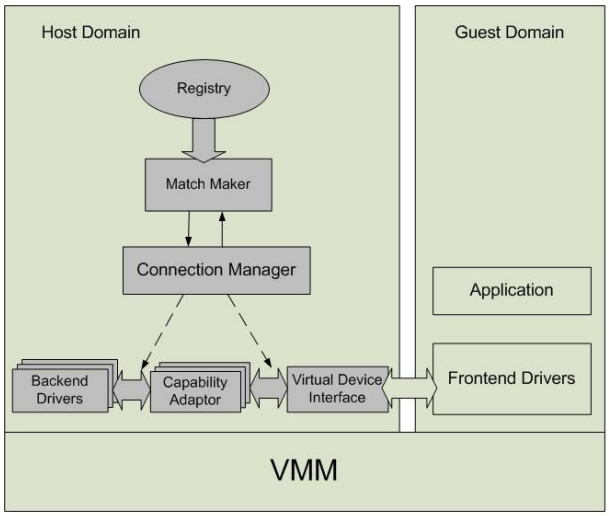

Figure 2: System Architecture

As shown in the figure, we have a pool of backend device drivers that issue commands to physical devices. Different than original Xen backend drivers, these drivers in our system don't 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 when the guest OS comes online. After the connection is made, the data flows without the involvement of the connection manager, as the arrows shown in the figure .

We also have a registry that keeps the 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 and make the connection manager aware of the appropriate adaptor modules.

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 (in Xen) unchanged and generic for the migration. The registry in our system allows the dynamic registration of capability adaptation algorithm, which can be used later for selection. Removing an entry in the registry prevents the 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.

5 Implementation

5.1 Discovery

Device discovery in Xen 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 wait for frontend drivers to discover. After it sees a frontend driver coming, the connection manager initiates an instance of virtual device interface and hand 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.

5.2 Connection establishment

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 for the connection to frontend drivers since we want to simulate the exact behavior of Xen's backend to make our system transparent to the frontend driver. 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 (described later) in order to make the connection happen successfully.

5.3 Capability adaptor interface

In our system, all capability adaptors need to implement certain functions in order to be able to be integrated into the system. An example is updateDisplay() function, which takes the original frame buffer and convert it into the desired size (e.g. one CA can convert 800*600 to 400*300 in updateDisplay whereas another CA can convert 400*300 to 800*600). We have a detailed list of functions that every CA has to implement. These functions help our system to install the CA dynamically at run-time.

5.4 User space driver

In our current implementation, we use a VNC-enabled backend device driver to simulate the physical device driver that actually controls the device. The backend driver starts a VNC server and executes the requests issued by the frontend in that VNC server to do frame buffer updates. Any VNC client can connect to the server to see the actual screen that is supposed to display on the physical device.

We use the simulated backend device driver instead of physical device driver for several reasons. First, the simulated driver can behavior exactly like the physical driver. The only difference is that instead of sending commands to physical device, it sends commands to the VNC server. Second, original Xen's virtual frame buffer backend driver is a similar VNC-enabled simulated driver. We can take advantage of that code to implement the VNC server and the updates. Third, by using a simulated driver, all the code are running at user level, which makes the development and debugging much easier. Note that Chameleon can still support both user space drivers as well as kernel space drivers.

6 Performance

We conduct an experiments to measure the overhead of the system compared to original Xen's performance. We use two identical AMD Athlon(TM) 64X2 dual-core machines with 1G memory and Ethernet connection to each other. Both machines have Fedora Core 5 Linux with Xen 3.1 installed as the software platform.

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 simulated VNC-enabled backend. We called this type of cost as "cursor blink". Figure 3 shows our results for different capability adaptors.

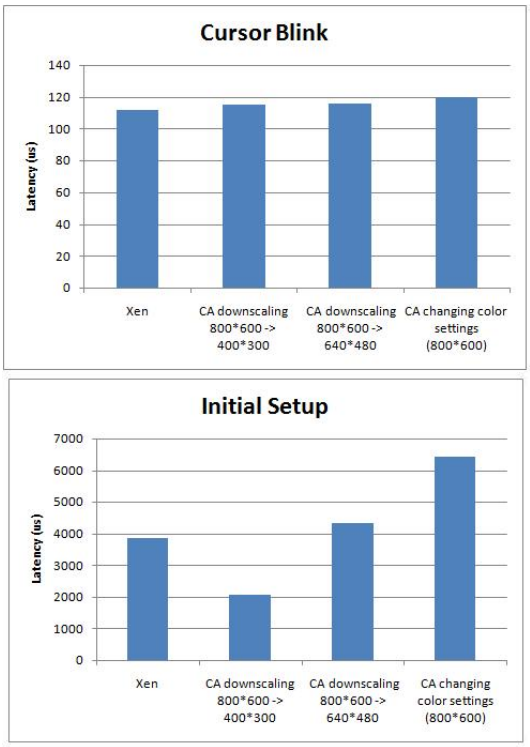

Figure 3: Performance of Chameleon and Original Xen

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 for big screen sizes. 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. 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. Note that for small screen size (400*300), our CA-enabled system even has better performance than original Xen's device driver (copy only a 400*300 screen instead of 800*600).

7 Related Work

The mechanism for adaptation can be provided at various layers of software stack, e.g. at the application level, middle-ware level or at the operating system/hypervisor level. In [1], Edmonds et al. discuss an application-specific adaptation model that places all the responsibilities with the application. The work by Becker et al. [3] presents a middleware approach that deals with device availability changes over time and location. W3C's *Composite Capability/Preference Profiles (CC/PP)* is predominantly used for proper representation of the hardware and software context in which an application is operating. Buchholz et al. [2] describe an alternate language named *Comprehensive Structured Context Profiles* for context representation. The primary objective of this work is to provide highly structured representation of context information for easing the task of dynamic composition(/decomposition) of context profiles. These approaches basically place the adaptation on top of operating system and are classified as application-level adaptation. Chameleon, however, deals with adaptation at operating system level and makes the application unaware of the change, which reduces the burden of the application programmer to consider the device level adaptations.

In the DCOD framework [5], 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. Naughton et al. in their work [4] discuss about a possible mechanism for providing dynamic modification of the Xen hypervisor. Their approach is based on Linux's loadable module mechanism which allows dynamic insertion and removal of kernel modules. In [8] Chen et al. have a slightly different objective: to make live updates to Linux kernels running in virtual machines. While these approaches are good ways to address key issues in their goals, they don't consider some state migration issues, which are essential when users move around (such as state migration). Chameleon, while enabling the dynamic installation of capability adaptors at system level, can also migrate device specific states along with the users. Therefore, Chameleon addresses migration issues more properly than the above system for mobile users.

8 Conclusions and Future Work

In this paper, we present our system, Chameleon, that adds the device migration and capability adaptation features into the current Xen virtualization system. Our system can allow dynamic selection of a variety of capability adaptors done at run-time and transparent to the guest domain. Through performance evaluation, we demonstrate our system introduces minimal overhead to the existing Xeno-Linux operating system.

Our future work includes the investigation of possible integration with middleware or application adaptation mechanism to make the migration more seamless. We can also collaborate with people in HCI to design more appropriate algorithms for our adaptation framework in addition to the existing general purpose algorithms. We will also seek to port our system on mobile platforms such as PDA or cell phone in order to support the migration and capability adaptation on those platforms. In addition, we plan to extend our system using device level virtualization technology to provide "virtualized" devices to upper layer operating systems in order to support further transparency for device migration.

References

[1] Edmonds, T.; Hodges, S.; Hopper, A., "Pervasive adaptation for mobile computing", 15th International Conference on Information Networking, 2001

[2] Buchholz, S. Hamann, T. Hubsch, G., "Comprehensive structured context profiles (CSCP): design and experiences", Second IEEE Annual Conference on Pervasive Computing and Communications Workshops, 2004

[3] Becker, C. Schiele, G., "Middleware and application adaptation requirements and their support in pervasive computing", 23rd International Conference on Distributed Computing Systems Workshops, 2003

[4] Thomas Naughton Geoffroy Vallee Stephen L. Scott, "Dynamic Adaptation using Xen", First Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007)

[5] R. Y. Fu, H. Su, J. C. Fletcher, W. Li, X. X. Liu, S. W. Zhao, and C. Y. Chi, "A framework for device capability on demand and virtual device user experience", http://www.research.ibm.com/journal/rd/485/fu.html

[6] Kientzle, Tim, "Scaling Bitmaps with Bresenham", C/C++ User's Journal, October 1995

[7] Smith, Alvy R., "A Pixel Is Not A Little Square!", Microsoft Technical Memo, July 17, 1995

[8] Chen, Haibo;Chen,Rong;Zhang,Fengzhe;Zang,Binyu andYew,Pen-Chung, "Live Updating Operating Systems Using Virtualization", The 2nd international conference on Virtual execution environments