VMedia: Enhanced Multimedia Services in Virtualized Systems

Himanshu Raj and Balasubramanian Seshasayee and Karsten Schwan

Georgia Institute of Technology, Atlanta, GA
{rhim,bala,schwan}@cc.gatech.edu

ABSTRACT

This paper presents the *VMedia* multimedia virtualization framework, for sharing media devices among multiple virtual machines (VMs). The framework provides *logical* media devices to virtual machines. These devices are exported via a well defined, higher level, multimedia access interface to the applications and operating system running in a virtual machine. By using semantically meaningful information, rather than low-level raw data, within the VMedia framework, efficient virtualization solutions can be created for physical devices shared by multiple virtual machines. Experimental results demonstrate that the base cost of virtual device access via VMedia is small compared to native physical device access, and in addition, that these costs scale well with an increasing number of guest VMs. Here, VMedia's MediaGraph abstraction is a key contributor, since it also allows the framework to support dynamic restructuring, in order to adapt device accesses to changing requirements. Finally, VMedia permits platforms to offer new and enhanced logical device functionality at lower costs than those achievable with alternative solutions.

1. INTRODUCTION

With their ever-increasing processing capabilities, even desktop-class platforms can now sustainably execute the workloads imposed by multiple concurrent applications. For instance, a single high end home PC's resources can be shared to simultaneously play a video game, watch a movie, and perform financial tasks. In this paper, we explore sharing opportunities and methods for multimedia devices, the goal being to make it easy to dynamically compose, share, and use these devices to provide efficient multimedia services. The concrete artifact resulting from this work is the VMedia addition to the Xen virtualization platform.[1] VMedia enables media-rich applications by better supporting flexible access to and use of the many media devices present in today's systems. Specifically, VMedia offers new hypervisor-level support both (1) for efficient and flexible device sharing and (2) for dealing with and exploiting device differences and diversity. Technical explanations of the VMedia approach and implementation appear next.

In virtualized systems, a workload can be run in its own isolated container, called a virtual machine (VM), each with its own runtime components, including operating and file systems. A Hypervisor (HV) or Virtual Machine Monitor (VMM), in conjunction with one or more privileged VMs, termed 'Service VM' or 'Domain 0' (Dom0), implement virtual instances of physical resources, such as CPU, memory, and I/O devices. Constituting a virtual platform for the VM, these virtual resources are multiplexed over the physical resources existing on the machine. Virtualization of basic resources like CPU and memory is implemented by the low-level hypervisor or VMM resident on the machine, whereas the Service VM is responsible for virtualizing devices. Using a Service VM allows ready utilization of device drivers for the multitude of I/O devices employed by VMs. For high end I/O devices, the functionality of the Service VM may also be provided directly by the device itself, in the form of self-virtualized devices.[2,3] VMedia is such a service VM based approach to virtualize multimedia devices.

Device virtualization is a key element of virtualized systems. A simple, non-intrusive method is to create a virtual device that emulates a physical one. In this case, the virtual platform provides I/O resources (configuration registers/memory) just like the physical platform, and the guest OS interacts with the virtual device in the same fashion as it did with the physical device, using its own device driver. However, this approach has inherent performance limitations, because device emulation requires fine-grain involvement from the HV and/or Service VM (i.e., at the level of memory/register access). As an alternative, all current system virtualization solutions provide simpler virtual I/O devices, which present different access interfaces to guest VMs, such as

shared memory circular buffer rings, rather than I/O memory and registers. Device drivers hide these interfaces from the guest OS kernel, providing it with standard device interfaces. For example, a virtual NIC device driver provides an ethernet interface that is identical to the interface provided by the physical NIC's device driver. Using these simpler virtual I/O devices and the corresponding device drivers provides substantial performance benefits compared to the emulation approach. Above this layer, guest operating systems, then, operate just like in non-virtualized environments, using their device drivers and other internal functionality to present applications with efficient higher level system abstractions like sockets, files, etc.

For modern virtualized platforms, then, researchers and developers have already recognized that such virtualization requires guest OSs to use new device interfaces and drivers. Several interesting research questions result from this fact, including (1) whether there are enhancements of such interfaces useful to certain classes of applications, and (2) whether such enhanced I/O devices can be implemented efficiently or even used to realize performance improvements?

This paper addresses these questions for multimedia systems and applications, by developing and experimenting with the *VMedia approach* to I/O virtualization. This approach exports to applications *logical devices* that are semantically enhanced versions of the physical devices present in the underlying platforms. Specifically, a VMedia logical device has attributes and provides access methods that go beyond defining "what the device is", as in current systems, to also define "how it is used". For example, a logical camera device might provide a rich multimedia access interface, like Video4Linux[4] (V4L), instead of the low-level API presented by a USB camera. Another example is an iSCSI-capable NIC,[5] which provides both a SCSI access interface for block devices and a normal ethernet access API.

Previous research has already demonstrated the utility of using logical rather than physical device interfaces. In our own work with V4L, for instance, we have shown that this interface can be used for transparent access to both local and remote physical camera devices.[6] The VMedia approach exploits I/O virtualization to go beyond such transparent device remoting: it provides a service-based interface to media devices in order (1) to allow sharing of these devices, and (2) make it possible to dynamically create from physical devices virtual ones with different properties and capabilities. We note here a similarity in approach between VMedia's service-based logical devices and modern file system services, such as NFS[7] and GPFS,[8] provided by today's network storage solutions. Such file services can be seen as a logical device, which are provided in addition to block-based virtual disk devices (e.g. devices supporting SCSI interface). Utilizing filesystem level abstraction, these storage logical devices allow sharing of content (files) in a straightforward manner, while the usual block-based virtual devices do not.

Previous work has also shown the utility of using semantic information to enhance certain physical devices, as with smart disks,[9] for instance. However, for cost reasons, these solutions have not been widely popular. VMedia addresses this issue by using software to enhance the virtual platform, rather than requiring new or extended device hardware (e.g., expensive device controllers). Furthermore, the service-based virtualization used by VMedia affords several additional advantages.

First, it can simplify the guest VM's OS kernel without sacrificing any of the functionality presented to applications. Second, by using domain-specific semantic knowledge, I/O virtualization at higher levels like V4L can provide better performance than solutions operating at the device level. Third, the use of logical devices can provide better opportunities for consolidation in the Service VM, based on information from multiple guest VMs. Fourth, a logical device may provide better performance and/or more functionality than that offered by a single physical device, by having the Service VM use an ensemble of physical devices to realize the logical device, for example. Shifting logical functionality to the Service VM also frees up computational resources at the guest VM side. A guest VM can then use these resources to implement other useful functionality. It may also increase platform's scalability in terms of the number of VMs it can support. Shifting computations related to I/O also allows guests to function better in the presence of resource restrictions, such as limited availability of cores or licensing restrictions imposed by software for certain number of cores.

In summary, this paper presents the VMedia framework for logical devices, focused on the multimedia domain:

- VMedia is used to export a 'multimedia' device to guest VMs, using the standard Video4Linux[4] interface.

- The multimedia device is implemented with software running in the Service VM, Dom0. By acting as a 'hub' for such logical virtual devices, the Service VM can provide enhanced multimedia services to guest VMs, with efficient and flexible device sharing, and offering new device capabilities.

Experimental results demonstrate the viability, utility, and performance of the VMedia approach and implementation. Multimedia device access can be performed by guest VMs via VMedia framework with low overhead (~$0.25ms$ for an image capture of size 320X240). Using semantic information, VMedia's low overhead virtualization solution allows multiple guest VMs to share a media device with minimal overhead (~$8ms$ for 8 VMs, each requesting images of size 320X240), as compared to the alternative method of semantic-unaware sharing via time-division sharing, which can impose overheads of upto 8X of physical device access cost for 8 VMs. We also demonstrate the ability to implement logical devices as aggregations of multiple physical devices, again with very low overheads (0.21%).

2. VMEDIA DESIGN AND ARCHITECTURE

VMedia Design. Unlike network and storage devices, which are virtualized via time- and space-sharing respectively, the rich semantics associated with multimedia devices make sharing at the device level more difficult. Web cameras and microphones, for instance, can be time-multiplexed among multiple VMs, but arbitration of the device will be difficult. For example, different VMs may want to change the attributes of the device in mutually exclusive ways. This means that the virtualization system must maintain a 'context' for the device per VM, and change the device to a particular context whenever the corresponding VM requests access. As a result, current virtualization solutions ensure that multimedia devices are used exclusively by one VM. Virtualization of these devices is done at a lower level, such as USB and PCI, and access is provided to a single VM as a passthrough.

The VMedia framework creates enhanced opportunities for sharing, by implementing logical devices that are accessed via a standard multimedia API, which is Video4Linux (V4L). Guest VMs' device drivers interact with the VMedia Service VM using a higher level API, again similar to V4L. VMedia's *virtual multimedia device* thus exported have several interesting properties. First, such a device need not be a simple mapping of the physical device that is being virtualized. In fact, additional interesting properties of a virtual device can be entirely implemented in software, an example being a virtual device that supports multiple palettes and image resolutions, while the physical camera supports only one of these. Second, device implementations can be entirely dynamic, using runtime code generation and extension techniques[10] and placing such extensions into Service VMs for shared use by all/some logical device users. Extensions may implement data transformations, for instance, to guarantee certain privacy constraints on the data captured by the device[6] or to provide data to end user applications in certain forms. Third and as explained next, multiple guests can efficiently share VMedia's multimedia devices, via its *MediaGraph* abstraction, described in detail in Section 2.3. The outcome is that a guest VM can be oblivious to how the physical device is being accessed, and that end users need not rely on complex applications hosted by guest VMs for such purposes.

The VMedia design also makes it possible to compose new virtual devices from multiple, possibly heterogeneous physical devices. For example, by using two similar camera devices and with appropriate phase lag, it is possible to support twice the frame rate than what could otherwise be afforded by a single device. As another example, a context sensitive camera device can be created using a camera and a microphone where an image is only captured in the presence of sound, else returning an image from a cache without capturing a new physical image.

VMedia Architecture The VMedia framework consists of two main components: (1) virtual multimedia devices and associated drivers running in guest VMs (client side), and (2) the VMedia runtime that executes in the Service VM, or Dom0 (server side). The VMedia runtime accesses the physical multimedia devices and provides guest VMs with access to the media data via virtual devices. Figure 1 depicts a high-level overview of these components.

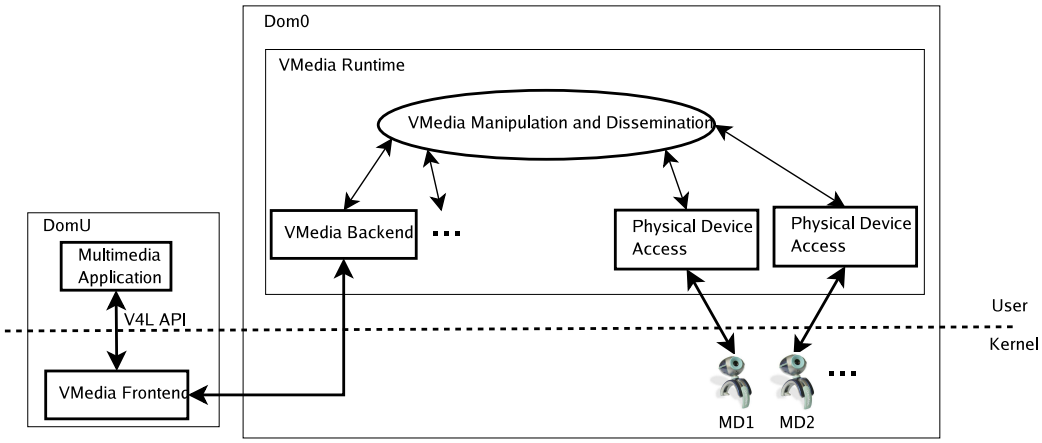


Figure 1. VMedia Architecture

2.1 Client Side Components

Client (Guest VM) side components include a virtual multimedia device and the corresponding kernel device driver. The virtual multimedia device is an extension of the virtual interface (VIF) abstraction presented previously.[2] The device is assigned a *unique ID* and consists of two message queues, each of which is a circular ring buffer. One message queue, called the *send queue* is for outgoing messages to the physical device, sent from the guest VM to the VMedia runtime. The other queue, called the *receive queue* is for incoming messages from the device, sent from the VMedia runtime to the guest VM. The simple API associated with these queues is as follows:

```
boolean isfull(send queue);
size_t send(send queue, message m);
boolean isempty(receive queue);
message recv(receive queue);
```

The functionality of this API is self-explanatory.

A pair of signals is associated with each queue. For the send queue, one signal is intended for use by the guest VM, to notify the VMedia runtime that the guest has enqueued a message in the send queue. The other signal is used by the VMedia runtime to notify the guest domain that it has received the message. The receive queue has signals similar to those of the send queue, except that the roles of guest VM and VMedia runtime are interchanged.

The kernel driver, called the VMedia frontend driver, registers a V4L device with the guest VM kernel. Applications running in the guest VM access the V4L device via V4L specific ioctls or file access system calls (e.g. read). These calls are converted into VMedia messages by the frontend driver, and sent to the other end via the send queue, where the backend component of the VMedia runtime receives them and performs appropriate actions. In response to these messages, the VMedia runtime may generate messages for guest VMs, which are received by the frontend via the receive queue. These messages in turn are mapped to application-specific calls. Table 1 shows the correspondence between key guest VM access API and VMedia messages. V4L-specific ioctls for VM API are capitalized. Also, VMedia messages in parentheses are receive queue messages, while others are send queue messages.

These messages do not carry media data themselves. All media I/O takes place via a pool of shared memory buffers shared between the guest VM and the VMedia runtime. These buffers can also be mapped directly in application address space, thereby allowing I/O with minimal copying.

The virtual devices we have implemented to date are those focused on the multimedia domain, supporting properties related to a video capture device, such as image size, image depth and palette, via the V4L interface. Properties for devices other than video, e.g. audio and VBI, can also be provided via this interface, and this is

VM API	VMedia Message
open	SETSIZE, SETPALETTE
read	GETFRAME, (RECVFRAME)
VIDIOCMCAPTURE	SETSIZE, SETPALETTE, GETFRAME
VIDIOCSYNC	(RECVFRAME)
VIDIOCSWIN	SETSIZE
VIDIOCSPICT	SETPALETTE

Table 1. Mapping between VM API and VMedia Messages

part of our future work. Virtual devices also support some VMedia-specific logical properties, such as orientation and quality, exported to applications via an extension of V4L API. These properties, along with the multimedia properties discussed above, are used by the VMedia framework to compose efficient and enhanced virtualized I/O solutions. Improved performance coupled with transparency to applications and to the guest VM's operating system are the potential outcomes of this approach, as shown in more detail in Section 4 below.

2.2 VMedia Runtime

The VMedia runtime realizes the self-virtualized I/O abstraction[2] with software resident in a Service VM. The runtime is responsible for:

- scalable and isolated multiplexing/demultiplexing of a large number of *virtual devices* mapped to one or more physical devices;
- providing a lightweight API to the hypervisor and guest VMs for managing virtual devices;
- efficiently interacting with guest VMs via simple APIs for accessing the virtual devices; and
- implementing multimedia domain-specific extensions that enable semantically enhanced logical virtual devices.

These functionalities can be broadly categorized as 'management' and as 'I/O virtualization'. For a virtual multimedia device, management functionality is provided to the hypervisor and to the guest VM using the device. In addition to obvious management actions like device creation and removal, the VMedia runtime provides additional, domain-specific reconfiguration functionality. For example, a video capture device may allow changes in image properties, such as colormap (color or grayscale), image depth and image size itself. The application running on the client side may request these changes, which in turn are sent to the VMedia runtime as management actions by the client side driver. The runtime makes appropriate changes in the properties associated with the virtual devices, along with any changes that may be necessary related to the I/O processing in order to satisfy these. For example, if the image size requested of a virtual multimedia device is different than that of physical device, an appropriate scaling filter may be installed in order to meet this mismatch. These reconfiguration actions are discussed in detail in Section 2.3.

The key functionality of the VMedia runtime is to implement I/O virtualization via sharing of physical multimedia devices among multiple virtual devices. The runtime utilizes semantic knowledge of virtual devices in order to perform this sharing. Since the runtime knows about the multimedia properties of the virtual device, e.g., the direction of I/O (input vs. output), type of content (such as image and audio), information about content (such as image size and colormap), it can use these properties in order to build an *information* flow from physical devices to virtual devices. For example, for input multimedia device, such as cameras, *images*, rather than *bytes*, are sent to virtual devices.

The VMedia runtime is composed of multiple entities that jointly realize the functionalities described above. These entities can be categorized broadly as (i) Physical Device Access, (ii) Virtual Device Backend, and (iii) Media Manipulation and Dissemination.

Physical device access entities implement the media device-specific methods for obtaining media data from or sending media data to the physical device, one entity per device. For example, the data could be obtained from

the USB based camera via a V4L-based device driver, or it could be obtained over the network if the camera is attached to a remote device, such as a cellphone connected to the host system via USB, bluetooth, or wireless. Depending on the type of device and how it is connected to the host system, the latency and throughput of media data will vary.

Corresponding to every guest VM frontend, the VMedia runtime contains a *backend* entity. These form a point-to-point connection with the frontend, and merely work as a gateway of information from (to) guest VMs to (from) VMedia runtime.

For input devices, such as cameras, captured media data from device access entities is provided to the VMedia manipulation and dissemination component (VMediaMD), where this data is transformed if required and is disseminated to the virtual device backend(s), which then flows to the guest VM frontend. For output devices, such as speakers, media data received from the guest VM is provided to the VMediaMD, where it is transformed if required and is provided to the appropriate physical device access entity for output. Currently, the VMedia framework only supports input devices, and hence, the remainder of this discussion is limited to these devices only.

The control flow for input multimedia devices (e.g., image capture requests and property changes) is similar to that of media data flow for output devices, with some exceptions. Management control requests may change the VMediaMD component itself. For example, if a virtual camera device requests a grayscale palette, the VMediaMD component may need to add another component to provide this functionality. Further, depending on the sharing of physical devices, if there is a common property/functionality required by all virtual devices and if it can be directly provided by the hardware, this control flow may reach the physical device access components themselves. Some of the management control decisions may only be taken at the service VM level itself, such as the orientation of the physical device.

I/O control requests go through a minimal path of the VMediaMD component, mostly providing arbitration. Arbitration decides which of the physical device access entities should receive this request (there may be more than one). VMedia-specific logical properties can also be used for arbitration. For example, a guest VM may indicate its preference for a certain viewing area via *orientation*. The arbitration logic matches this preference to one or more physical devices. Arbitration also decides whether it is necessary to forward a request to a physical device, since it may already be involved in the I/O. The request is only forwarded if it is not.

Media sharing in VMedia is governed by a simple arbitration principle – any request received from the guest VM during the time when a media capture I/O is pending can be satisfied from the result of this capture. Hence, if multiple VMs issue capture requests simultaneously, the capture is performed only once and the result is distributed to all VMs. This type of device sharing is a special case of space sharing, where a device can be shared by all virtual devices at all times due to the *semantic* properties of the device.

2.3 The MediaGraph Abstraction

Abstractly, the VMedia runtime entities described above and the control and data flows implemented by them form a di-graph structure, termed MediaGraph. This graph is built to meet the properties specified by end user applications for the virtual multimedia devices they are using. Specifically, the MediaGraph implements efficient media dissemination by consolidating common computations and by reducing communication costs via data filtering. Moreover, the MediaGraph abstraction is dynamic – it can be modified when new virtual devices are added and/or when the properties of existing multimedia devices are modified. Such modifications are triggered by configuration events generated by guest VMs and/or by monitored changes to devices.

Physical device access entities and virtual device backends form the edge vertices of the MediaGraph (sources and sinks, respectively), whereas VMediaMD entities form the internal vertices. These internal vertices correspond to various arbitration and transformation functions. Transformation functions perform the necessary conversions from the media format provided by the physical sources to formats desired by the backend at the guest VMs, and directed edges in the MediaGraph represent the control and data flows.

A sample MediaGraph is shown in Figure 2. As seen in the figure, the cameras, represented by the source nodes S_1 and S_2, generate image frames, that are then sent to the transformation nodes T_1 through T_4, that

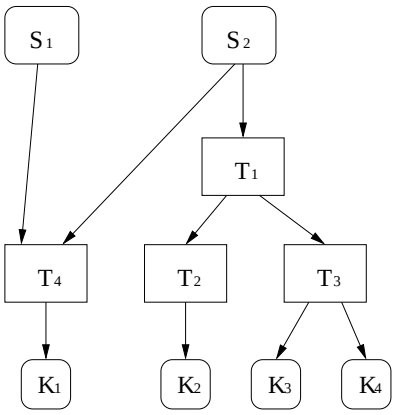


Figure 2. An Example MediaGraph

perform transform operations on the images and send the final outputs to the backend nodes K_1 through K_4, which provide the processed images to the multimedia guest VMs.

The MediaGraph abstraction enables efficient sharing of the multimedia content by avoiding redundant transformations that may be required by multiple sink nodes (backends) in order to support the properties. For instance, the graph shown in Figure 2 combines the common transformation T_1 for backend K_2, K_3 and K_4, thereby reducing the overall cost paid by the VMedia runtime. Next we describe an algorithm to maintain the efficient sharing when a sink node is added or deleted for a MediaGraph containing single source node.

Maintaining the MediaGraph for Efficient Sharing of a Single Source The amount of computation required to carry out the transformations in the MediaGraph is dependent on the topology of the graph. For instance, if the source image needs to be transformed into a grayscaled and scaled down image in order to serve a VM, performing the grayscaling prior to the scaling down operation involves extra computation than performing the other way round (since the grayscaling is applied on a larger image). Further, the structure of the MediaGraph is dynamic, due to addition/deletion of sinks, and also due to changing content parameters (e.g., resolution, color depth, etc.). In this work, we use a greedy algorithm to build and maintain the MediaGraph, in response to changes in the sinks.

To select the sequence of transformations to be performed in the graph, we group its transformation actions along the various parameters that define the content, and further define an ordering relation among the parameters within each group. For any two parameters a and b within a group, we define the ordering relation $a < b$ if the information content in b is more than that in a. Using this relation, for instance, we can define the depth of the image with the ordering 8bpp $<$ 16bpp $<$ 32bpp. The groups themselves are ordered in the graph in such a manner that data reducing operations (like scaling down, for instance) are closer to the sources than data increasing operations, in order to minimize the computation costs. The multimedia content at any node can thus be represented using the n-tuple $< a_1, a_2, ..., a_n >$, with each tuple corresponding to a group, and $a_1, a_2, etc.$ each represents the parameter within its group. As discussed previously, the groups themselves are ordered in non-decreasing order of computational complexity, to minimize the amount of processing carried out in the transformations. An example 3-tuple is <resolution, depth, grayscale/color>.

When a virtual camera is opened by a process in a guest VM, this action is translated to an addition of a sink corresponding to the VM, to the MediaGraph. Conversely, when the virtual camera is closed by the process, or if the guest VM is destroyed, the corresponding sinks are deleted from the graph. On the addition (Algorithm 1) of a sink (with desired content parameters $< r_1, r_2, ..., r_n >$) to a source node with parameters $< s_1, s_2, ..., s_n >$, a check is performed to see if any of the desired parameters exceed the source parameters (line 2), and if so, the source parameters are updated to the maximum of the existing and the desired parameters (line 3), with all other transformations updated accordingly (lines 4-7). Next, the graph is traversed starting from the source node to find a maximal match of the desired parameters among those of the existing transformations (lines 9-12), and finally the sink is connected to this node via necessary transformations (lines 13-20).

Algorithm 1 Addition of a sink

1: Let the sink's required parameters be $< r_1, r_2, ..., r_n >$, and the source parameters be $< s_1, s_2, ..., s_n >$.
2: **if** $\exists k, 1 \leq k \leq n$, such that $r_k > s_k$ **then**
3: Change the source's content to match $< max(r_1, s_1), max(r_2, s_2), ..., max(r_n, s_n) >$, where $max(a, b) = a$, if $a > b$ and b otherwise
4: **for all** $j, 1 \leq j \leq n$ **do**
5: $s_j \leftarrow max(r_j, s_j)$
6: $\forall$ transformation node N if $input(N) \neq s_j$, change the transformation so that $input(N) = s_j$
7: **end for**
8: **end if**
9: $N \leftarrow sourcenode$, $j \leftarrow 0$ {Now, $\forall k, 1 \leq k \leq n, r_k \leq s_k$}
10: **repeat**
11: $parentnode \leftarrow N$, $j \leftarrow j + 1$
12: **until** $\forall N$, such that $parent(N) = parentnode$, $output(N) \neq r_j, \forall j, 1 \leq j \leq n$
13: $currentnode \leftarrow parentnode$
14: **for all** $j, 1 \leq j \leq n$ **do**
15: **if** $r_j < s_j$ **then**
16: Create a transformation node T, such that $input(T) = s_j$, $output(T) = r_j$
17: Connect $currentnode$ to T, such that $parent(T) = currentnode$
18: $currentnode \leftarrow T$
19: **end if**
20: **end for**
21: Connect sink to $currentnode$

Deletion of a sink (Algorithm 2) begins with removing the sink node from its parent (line 1), and then traversing towards the source node until all unnecessary transformation nodes (those that serve no other nodes) are removed (lines 2-3). Finally, it is determined if the source's parameters can be lowered due to the removal of the sink, (lines 5), and if so, carried out (lines 6-11).

Changing a sink's parameters results in the actions of the deletion of the sink node from the MediaGraph, followed by an addition of a sink node with the new parameters.

Algorithm 2 Deletion of a sink

1: Disconnect sink from $parent(sink)$, set $currentnode \leftarrow parent(sink)$
2: **while** $|children(currentnode)| = 0$ **do**
3: Delete currentnode, set $currentnode \leftarrow parent(currentnode)$
4: **end while**
5: **if** $currentnode = sourcenode$ **then**
6: **while** $|children(sourcenode)| = 1$ and the child is not a sink node **do**
7: Set $N \leftarrow child(sourcenode)$
8: Change source's parameter s_j to $output(N)$ for appropriate j
9: Delete node N
10: $\forall T$, such that $parent(T) = N$, connect sourcenode to T
11: **end while**
12: **end if**

3. IMPLEMENTATION DETAILS

The VMedia runtime is implemented as a user space application in the Service VM (Dom0) which completely encapsulates the I/O virtualization for the multimedia devices. Backend entities communicate with the frontends in guest VMs via Xen HV-specific communication mechanisms, which provide for the shared message queues and signaling. Different physical device access entities are run as separate threads to provide maximize concurrency

in the runtime. These threads use device-specific methods for I/O. For example, for a USB based camera, the corresponding thread uses V4L ioctl calls for image capture, similar to applications such as camE.[11] For a cellphone based camera, the corresponding threads communicates with a server process running on the cellphone that provides images over network.

For information dissemination between these edge nodes of the graph and to implement VMediaMD entities, the runtime utilizes an event-driven middleware, called EVPath.[12] EVPath allows data flow as *events* among nodes of an overlay termed *stones*. Stones can perform event processing, and can can transform an input event to an output event, possibly of different type, before passing it on to another stone. Stones can also perform routing decisions based on the event contents. This allows EVPath to perform content adaptation, which is required to support the logical functionality provide by VMedia.

The VMedia framework allows two types of logical functionality – one encoded in the V4L attributes of the virtual device itself, e.g. image size and colormap, the other completely based on guest VM. In the former case, the VMedia runtime installs well-defined processing entities as stones in the MediaGraph. For example, if the image size of a virtual device is smaller than the physical device, a stone containing a scaledown filter is installed in the MediaGraph. Other filters, such as crop and grayscale, are installed in a similar fashion. VMedia also allows further predefined logical functionality via the extension of V4L attributes. For example, a virtual camera device may provide image data in specific image formats, such as JPEG and PNG. These functionalities can be provided in a manner similar to the earlier ones. These image processing-specific functionalities are implemented using the imlib[13] library.

4. EXPERIMENTAL EVALUATION

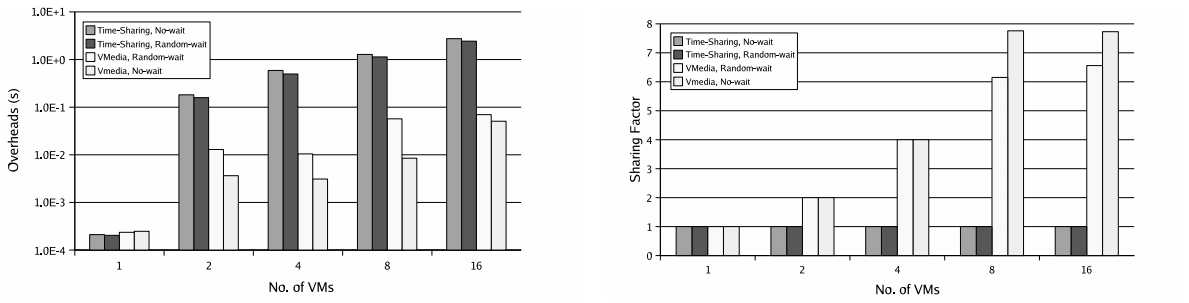
We evaluate the VMedia framework on a desktop system with 3.2GHz dual-core Pentium-D processor and 3GB of RAM. To this machine are attached a Kensington SE401 USB-based camera, and a second Motorola e680 cellphone with a built-in camera. The e680 cellphone runs the Linux 2.4.20 kernel and is connected to the desktop via USB. The camera communicates with the desktop using the TCP/IP protocol, supported by a virtual network driver over USB stack. Service VM (Dom0) running VMedia runtime is allocated 512 MB RAM and one of the physical CPUs, and runs the Linux 2.6.16 kernel. Other CPU is shared among guest VMs, as determined by Xen's scheduling policy. The VMM virtualizing the desktop system is Xen version 3.0.3.

4.1 Overheads of VMedia Framework

This set of experiments quantifies the overhead of multimedia virtualization via the VMedia framework, measured as the difference between the latency of image capture experienced by a guest VM from the virtual multimedia device and the latency of image capture experienced by the VMedia framework from the physical device. This overhead includes the cost of transformations performed on the media data (computation), and its dissemination to virtual device frontends (communication). The content is delivered only to those virtual devices that request it, even if these virtual devices share some (or all) of the VMediaMD components with other devices that did not request it. For these experiments, the image properties (size, palette etc.) for virtual and physical media devices are kept the same, so the only overhead incurred is due to dissemination.

The scalability of VMedia is demonstrated by increasing the number of VMs and measuring the *amortized overhead*. As number of VMs are increased, transmission costs of VMedia runtime increase as media data needs to be disseminated to more and more VMs. However, the latency of image capture as experienced by a guest VM depends on the amount of sharing, as the cost of a physical I/O gets amortized over multiple virtual I/O requests. To capture this sharing effect, we only account for the net positive overhead experienced by a guest VM, which includes VMedia's dissemination cost. We average over all net positive overheads experienced by N VMs sharing a physical device, and report it as amortized overhead. The overall cost of virtualization also increases due to scheduling, since context switching of VMs is required on a single CPU. In future multi- and many-core systems, the scheduling costs will be smaller, or even negligible, if there are enough physical CPUs.

We compare the VMedia overhead with a *time-sharing* approach of virtualizing the multimedia device. In this approach, every guest VM image capture request results in a image capture from physical device. In the presence of no contention, this approach is comparable to VMedia. However, in case when multiple VMs require



(a) Overhead, results are reported on log scale on y-axis. (b) Sharing factor.

Figure 3. Comparative evaluation of VMedia and Time-sharing Approach.

access to the media device, the overhead of this approach not only includes communication of media data from the service VM, it may also include image capture latency from physical device for another VM. The overhead of this approach, hence, is always positive, and we report the average overhead per request.

We evaluate both VMedia and time-sharing approaches in two scenarios. In one scenario, termed 'no-wait', VMs successively request image capture from virtual devices without any wait between them. In another scenario, termed 'random-wait', a VM waits a random amount of time between $[0, 1]$ seconds before making another request.

Figure 3(a) compares the overheads of VMedia and time-sharing approaches. For a single VM, the overhead of both the approaches are negligible. However, as the number of VMs increase, the overhead of time-sharing approach increases rapidly, including multiples of physical capture time as a factor, in both 'no-wait' and 'random-wait', with latter being slightly better than the former. The overhead of VMedia approach also increases, but only due to the communication cost of VMedia and context-switching cost of VMs. Both of these overheads are small when compared to the physical capture time. The overall overhead of I/O for an image capture from the virtual device with increasing number of guest VMs becomes as high as ˜25% of the overall virtual device capture cost.

For each scenario, we also present the *sharing factor*, which demonstrates the underlying approach's ability to share the device, the higher the better. This factor is calculated as $\frac{\sum_{i=1}^{N} \text{captures from virtual device i}}{\text{captures from physical device}}$, N being the number of guest VMs. Figure 3(b) compares the sharing factor for different virtualization approaches in two scenarios, as mentioned earlier. For perfect sharing, the sharing factor should increase linearly with increasing number of guest VMs. However, due to high context switching costs, we observe the best case sharing factor to be ˜8. The sharing factor of time-sharing approach is always 1, since every virtual capture request results in a physical capture request. The VMedia approach attains best sharing factor for the 'no-wait' case, while the sharing factor reduces as the contention for the physical device is reduced in the 'random-wait' case.

These results show that the VMedia framework shares physical devices efficiently, which in turn contributes to its performance and scalability. Further, using higher level 'V4L' requests, the no-sharing passthrough type virtualization for a single VM can be achieved at a lower cost than, e.g., using USB level requests[14] – where every single USB level request adds an overhead of about 25%.

4.2 Enhanced functionality sharing

Results in the previous section demonstrate the performance benefits derived from device sharing and the consequent amortization of I/O costs. However, using MediaGraph, the VMedia framework affords further benefits by sharing at the *logical* level. To demonstrate the benefits of enhanced sharing via the MediaGraph, we construct the following scenario. Four guest VMs are created in Xen – two VMs, VM1 and VM2, require images of size 640x480, while VM3 and VM4 require images of size 160x120. VM4 also requires grayscale images. We compare two approaches to sharing – the naive way, where the VMedia framework only shares the physical device and any transformations are performed by the guest VMs themselves and the enhanced way, where the VMedia

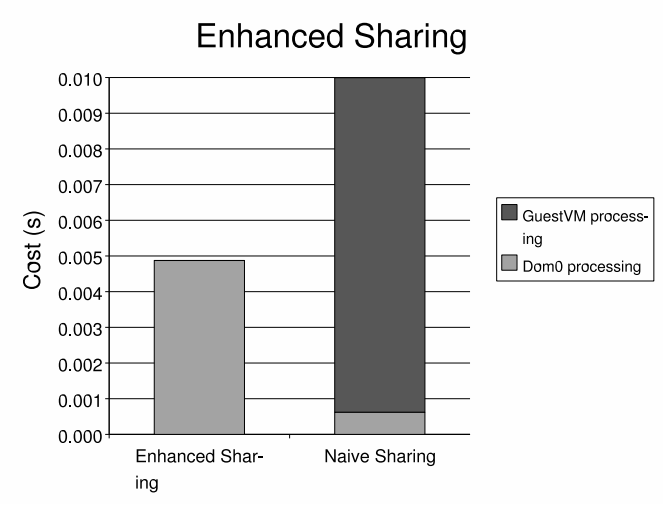


Figure 4. Enhanced Sharing

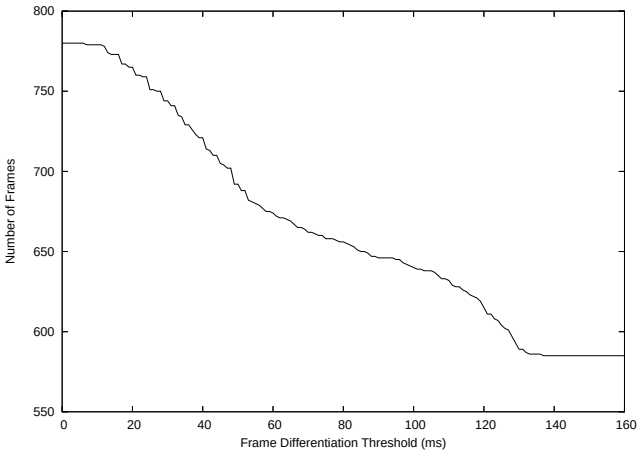


Figure 5. Number of Distinct Frames from two cameras in Response to Changing Frame Differentiation Threshold.

framework also performs any required transformations. These transformations are derived by the framework based on the *parameters* of the virtual devices, namely image size and color palette.

In this case, since the MediaGraph reduces the amount of redundant transformations performed, we expect to see lower processing costs. As shown in Figure 4, since all transformation-related processing is performed in the Service VM with the MediaGraph, we see a higher cost. In the naive sharing case, the images are simply sent to all VMs. However, the guest VMs perform all of the transformations in the latter, which is completely absent in the former. The overall costs, as shown in the figure, are almost 50% lower due to elimination of redundant computation.

4.3 Dynamic Restructuring of MediaGraph

In this section, we quantify the overheads of VMedia framework associated with dynamic restructuring of MediaGraph. Restructuring is performed in response to the management actions performed on the virtual media devices. These actions include opening and closing of devices, and changing their properties, such as image size and color palette, via ioctl calls. The framework translates these actions into MediaGraph modifications, as described earlier in Section 2.3. The modifications require creation and removal of nodes from the graph, which in turn require EVpath *stones* to be added/removed.

We measure the cost associated with a management action as 1) the time it takes VMedia runtime running in dom0 to carry out the modifications, and 2) the amount of change in the MediaGraph resulting from these modifications. Since the removal of stones takes significantly less time compared to their additions, we only consider the number of stone additions as the metric for the amount of change in the MediaGraph. Figure 6 depicts these results. On x-axis, we vary the number of VMs (and hence the number of virtual devices). Each VM performs 100 management actions related to device property changes. Each action is drawn randomly-uniformly from a set of 5 such changes - 3 related to image size changes and 2 related to color palette changes. Each VM also waits for a random amount between $[0, 1000)$ milliseconds, between two consecutive actions.

Results demonstrate that with increasing number of VMs, the average cost per action decreases, since the cost of MediaGraph change could be amortized over actions from different VMs. Also, the amount of change required for MediaGraph increases sub-linearly. Put differently, the amount of change per management action decreases with increasing number of VMs. This explains the decreasing average cost of management actions.

4.4 Enhanced Logical Devices via Multi-Device Aggregation

Depending on the requirements of a guest VM and the availability of physical devices, certain services can be composed that allow a guest VM *improved* quality of service. For example, if a guest requests a wide image (of aspect other than regular 4:3), VMedia can aggregate images from multiple cameras either horizontally and/or

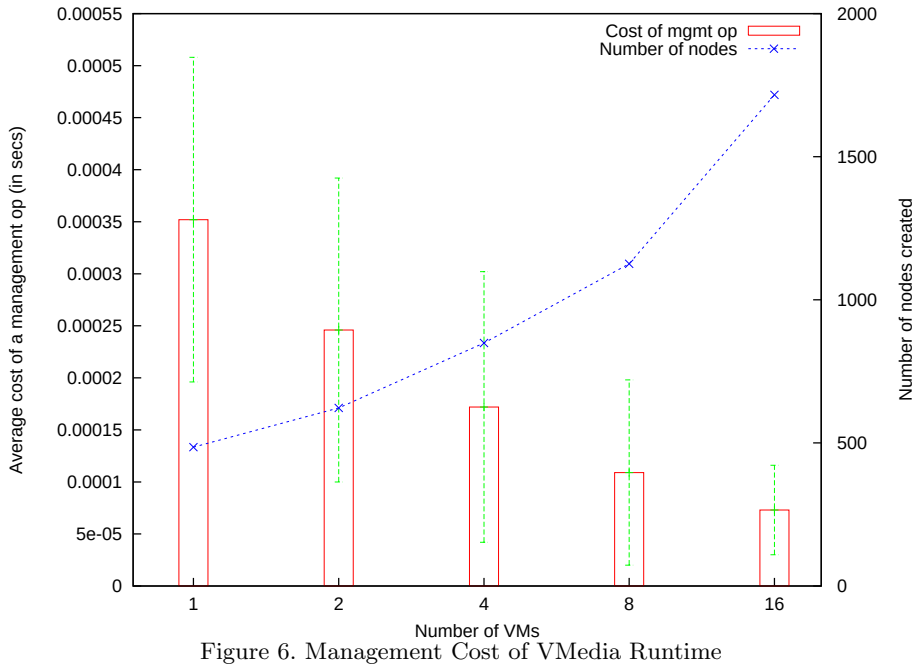


Figure 6. Management Cost of VMedia Runtime

	Cost (ms)	Cost (% of Vdev Cost)
Vdev Capture	622.023	100
Phys Capture	619.624	99.61
Transformation	.907	0.15
Communication	.366	0.06
Miscellaneous	1.127	0.18

Table 2. Cost components for Multi-Camera Aggregation via Concatenation

vertically. Similarly, if a better resolution image is required, e.g. 640X480, but physical cameras can only provide 320X240, four such cameras can be aggregated. This is better than just using scaling – since it will not result in any quality loss. This can be further extended with additional processing to create a video wall.[15] Table 2 shows the microbenchmarks for a virtual camera device created by the concatenation of two cameras, the USB camera and the cellphone camera. The cost of physical device capture is taken as the maximum of these two devices, which corresponds to the cellphone camera. VMedia framework overhead includes the concatenation transformation action and the communication cost, and is very small compared to I/O latency.

Another example of aggregation is to use multiple media capture devices, possibly with a phase lag, in order to minimize the average latency of media access to guest VMs, where these devices are sampling the same environment. For example, for a single continuous image source with interframe latency L, average latency for capturing an image is $L/2$ – assuming accesses arrive randomly over a uniform distribution. However, by using two such image sources, and running then with phase lag $L/2$, the average latency can be reduced to $L/4$, effectively doubling the frame rate. To demonstrate the viability of this approach, we use two cameras, one USB and one cellphone camera, to capture frames in parallel, in a time period T, and timestamp them. The latency of frame capture from USB camera is ˜200ms, while from cellphone camera, it is ˜600ms (˜300ms of which is the core physical capture latency on cellphone and rest of it is the network transfer over USB to desktop machine.) Next, we coalesce i'th frame from device 1 ($f_{i,1}$) and j'th frame from device 2 ($f_{j,2}$), iff $|T_{f_{i,1}} - T_{f_{j,2}}| < \delta$, where δ is the frame differentiation threshold. This threshold quantifies the difference in media content, and hence the value, provided by successive frames captured by different devices. At lower values of frame differentiation threshold, the added value of extra frame is less. The resultant number of frames denote the *valuable* content. We plot the resultant number of frames obtained in a 2 minute time-period against different values of δ, as shown in Figure 5. The result shows that the aggregate device can achieve more distinct frames than a single camera,

and hence can provide better frame rate to the clients. Note that for high values of δ, the number of distinct frames are small, and asymptotically reach the number of frames provided by the faster device (~580 in this case), thereby limiting the benefits from using multiple devices. For lower values of δ, the number of frames are larger, although the difference in media content may be smaller, again limiting the benefits from using multiple devices.

Alternatively, such services can be created in the guest VM itself – if we provide one virtual media device per physical device. This can be accomplished, e.g., by the VMedia framework itself, by creating multiple MediaGraphs, one for each physical camera. The passthrough access to physical devices can be utilized in a similar fashion. As argued earlier, the latter approach does not provide sharing, and hence is of little interest. We believe that a single MediaGraph with support for aggregation is better than aggregation in guest VMs, for the following reasons:

- The guest VM implementation couples virtual devices with the physical environment, e.g., in number of devices and their orientation. This is usually a concern in virtualized environments, since a VM may be migrated to a different physical platform. Hence, the service implementation on guest VMs must be able to adapt to any changes in the physical platform. This adds complexity for VMs. By keeping this functionality purely in the VMedia framework, the framework – local to a single physical platform – provides a better way to provide this service.
- A single MediaGraph allows for enhanced sharing, in case aggregation is utilized by multiple VMs. Computations for logical functionality can be performed once, and results can be shared among multiple guest VMs.

5. RELATED WORK

Efficient methods to virtualize basic system resources like CPU and memory have been well studied. Recent efforts in virtualization have focused on efficient sharing of I/O devices such as network interface.[2] The VMGL[16] approach virtualizes a video card to provide hardware-based 3D acceleration to guest VMs. As identified in the paper, standardized higher level interfaces improve both the ease of implementation and the adoption of such solutions. VMGL uses the OpenGL abstraction as the interface, whereas the VMedia framework uses the Video4Linux[4] interface.

Aggregating multiple devices to provide richer services has been studied along several dimensions. Superimposed projection[17] discusses fundamental issues arising when using multiple projectors to produce a single high-resolution image. The Princeton scalable display wall project[15] also discusses algorithms to solve alignment, color balancing, and other problems arising in a distributed environment. The Lyra system[18] studies timing services that can be provided to multimedia applications, for achieving better quality of service. Such services can be harnessed in multimedia scheduling in a virtualized environment to provide QoS guarantees to guest VMs, as well as to schedule fine-grained captures in frame aggregation (Section 4.4) for example.

Research focusing on sharing multimedia include the Irisnet project,[19] which applies filtering on distributed multimedia sensors to deliver customized content. Feeds from several remote webcams connected to the Internet are used to compose useful content and services built on top. MSODA[20] proposes a multimedia service overlay among virtual machines for media service access and composition. VMedia framework focuses on providing multimedia services to virtual machines via higher level 'logical' devices, while services are implemented in the Service VM. The Indiva middleware[21] also provides a higher level, file system abstraction, for composing distributed multimedia content.

6. CONCLUSIONS AND FUTURE WORK

Efficient multimedia device virtualization and sharing requires that these devices be virtualized at a higher, 'semantic' level, rather than the traditional approaches, which incur high overheads. This can be obtained via a service-oriented approach to virtualization, where logical devices share semantic knowledge with the Service VM that virtualizes the device. The VMedia framework implements such an approach, for virtualizing multimedia

devices among multiple guest VMs. The framework also allows further benefits via enhanced functionality sharing, and it can potentially reduce the overall cost of multimedia services provided to guest VMs. Experimental results demonstrate that the overhead of the VMedia framework is small, and that it scales well with increasing numbers of virtual devices and virtual machines. The framework also supports dynamic adaptation in response to the changing demands of guest VMs, communicated in terms of virtual device properties and guest specific computations, and enhanced virtual devices with new and interesting functionalities via aggregation of multiple physical devices.

In the future, we plan to extend the VMedia framework to include devices from multiple systems in a distributed environment. This requires that the MediaGraph composition and restructuring span multiple nodes. The VMedia framework can also be extended to include other types of devices, such as sensors and storage devices. Using multiple heterogeneous types of devices provides opportunities for interesting functionality that could be exported to guest VMs by the Service VM. One such example is *context-aware storage*, where context is derived from media devices, and based on that, access of certain content is performed from a specific storage device.

REFERENCES

1. B. Dragovic *et al.*, "Xen and the Art of Virtualization," in *Proc. of SOSP*, 2003.
2. H. Raj and K. Schwan, "High Performance and Scalable I/O Virtualization via Self-Virtualized Devices," in *Proc. of HPDC*, 2007.
3. J. Liu, W. Huang, B. Abali, and D. K. Panda, "High Performance VMM-Bypass I/O in Virtual Machines," in *Proc. of USENIX ATC*, 2006.
4. "Video4linux resources." `http://www.exploits.org/v4l`.
5. "Internet small computer systems interface." RFC 3720.
6. J. Kong, I. Ganev, K. Schwan, and P. Widener, "CameraCast: Flexible Access to Remote Video Sensors," in *Proc. of MMCN*, 2007.
7. "Network File System version 4 Protocol." RFC 3530.
8. F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proc. of FAST*, 2002.
9. M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Semantically-Smart Disk Systems," in *Proc. of FAST*, 2003.
10. I. Ganev, *A Pliable Hybrid Architecture for Run-time Kernel Adaptation*. PhD thesis, College of Computing, Georgia Institute of Technology, 2007.
11. "CamE." `http://directory.fsf.org/camE.html`.
12. "EVPath." `http://www.cc.gatech.edu/systems/projects/EVPath/`.
13. "imlib." `http://freshmeat.net/projects/imlib/`.
14. S. Kumar, S. Agarwala, and K. Schwan, "Netbus: A Transparent Mechanism for Remote Device Access in Virtualized Systems," Tech. Rep. GIT-CERCS-07-08, Georgia Tech, 2008.
15. G. Wallace *et al.*, "Tools and Applications for Large-Scale Display Walls," *IEEE Computer Graphics and Applications* , July 2005.
16. H. A. Lagar-Cavilla *et al.*, "VMM-Independent Graphics Acceleration," in *Proc. of VEE*, 2007.
17. N. Damera-Venkata and N. L. Chang, "Realizing Super-Resolution with Superimposed Projection," in *Proceedings of IEEE International Workshop on Projector-Camera Systems*, 2007.
18. C.-W. Yang, P. C. H. Lee, and R.-C. Chang, "Lyra: A System Framework in Supporting Multimedia Applications," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, 1999.
19. P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan, "IrisNet: An Architecture for a World-Wide Sensor Web," *IEEE Pervasive Computing* , October-December 2003.
20. D. Xu and X. Jiang, "Towards an Integrated Multimedia Service Hosting Overlay," in *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, 2004.
21. W. T. Ooi, P. Pletcher, and L. A. Rowe, "Indiva: A Middleware for Managing Distributed Media Environment," in *Proc. of MMCN*, 2004.