# Creating Efficient Execution Platforms with V(irtualized)Services [⋆]

Balasubramanian Seshasayee and Karsten Schwan

Center for Experimental Research in Computer Systems, College of Computing
Georgia Institute of Technology, Atlanta, GA 30332.
{bala, schwan}@cc.gatech.edu

**Abstract.** Virtualization is creating new opportunities for innovative uses of middleware technologies. In this paper, we present such opportunities for mobile or pervasive environments, where virtualization efforts face challenges that include the diversity of platforms and devices present in these domains, dynamic device and resource behaviors, and the efficient use and sharing of such resources. Specifically, we leverage middleware to significantly extend virtualization technologies in terms of their efficient support for resource sharing in the presence of diversity, dynamics, and mobility. The outcome is what we term the *Virtualized Services* (VServices) approach to representing and using devices and resources. VServices extend the virtual device interfaces used in existing virtualization infrastructures to go beyond sharing physical devices among multiple virtual machines, to also sharing logical entities that internally use middleware to provide new services to end user applications. By using and sharing the higher level VServices abstractions instead of physical devices, opportunities are created (i) to optimize the implementations of certain services without requiring changes to VM implementations, (ii) to enhance device functionalities by combining software service implementations with the physical devices being used, again without changing the basic nature of VM-device interactions, and (iii) to emulate devices whose physical realizations may be remote or non-existent. Experimental evaluations conducted on an implementation of these concepts in the Xen virtualization infrastructure exhibit up to 50% improvements in latency as well as improved performance scalability, when compared to current Xen implementations of such devices. This paper also describes practical realization of device enhancements using VServices.

## 1 Introduction

While virtualization technologies like those provided by Xen or VMWare have been shown widely useful for server systems, efforts now underway in industry and academia are exploring their utility for mobile and ubiquitous computing. This is because of their ability to offer improved serviceability (e.g., upgrades)

⋆ This work was funded in part by a grant from Motorola Labs and an NSF-ITR award.

and maintainability (e.g., recovery) through multiple isolated execution domains, termed Virtual Machines (VMs), and because they can deal with dynamic platforms and environments through methods like runtime VM [1] and device migration [2].

Our research is taking the further step of exploring the use of virtualization for presenting to guest VMs and their applications a *virtualized platform* that provides them with a uniform execution environment that is largely independent of actual physical hosts and resources. The practical challenge, of course, is to present such an environment without requiring undue changes to guest VMs and applications. To attain this goal, our work leverages modern middleware technologies to create *Virtualized Services* (VServices) that enhance the execution platforms provided by current hypervisors (or Virtual Machine Monitors, VMMs) like Xen [3] or VMWare [4]. Since these enhancements provide to guest VMs the *services* they require, at a higher level of abstraction than the device-level virtualization currently performed by VMMs, opportunities are created to introduce optimizations to improve resource utilization and access and/or to share services so as to remove redundancies in their use. In addition, new or enhanced devices and device capabilities can be supported, without introducing additional complexities into guest VMs. Similarly, service realizations can be based on physical devices, on software enhancements of such devices [5], or on software device emulations [6]. Further, remote service access, even with mobility, can be implemented with any or multiple of the many existing middleware-based access methods, without requiring guests to adopt specific middleware solutions (e.g., .NET vs. Java). Finally, general middleware paradigms such as tuple spaces [7], publish-subscribe [8], etc., as well as middleware techniques designed for mobile solutions, surveyed in [9] and [10], can result in future service realizations that present even higher level abstractions to guest VMs and their applications, sometimes termed 'application virtualization' [11].

Our Xen-based implementation of VServices provides service realizations that combine software with hardware solutions to provide new functionality to end user applications. For instance, a simple web camera is enhanced to emulate a TV tuner card, by combining it with a software service implementing publish-subscribe access methods. A second example is a VService presenting a 'non-existent' device to a guest VM, by emulating a non-existent GPS device implemented either with bluetooth-based or external camera-based indoor localization methods [12], depending on the external environment and device availability. With VServices, we then show that existing applications can use them without requiring any modifications. We also explore the opportunities offered by VServices for service consolidation and quality of service management.

Basic properties of VServices demonstrated in this paper include the following:

- low overhead of service access and activation,
- high performance in service execution, and
- opportunities for service consolidation leading to improved efficiency in resource use.

Given these results, the principal technical contribution of this paper, then, is a demonstration of the important role modern middleware technologies can play in creating uniform and consistent execution platforms for guest VMs and applications. This is attained by packaging such enabling middleware into its own, isolated execution containers, deployable and changeable independently from guest VMs' operating systems and application, and which use software methods that need not depend on the operating systems or execution environments used by VMs. The VServices approach exposes these containers to applications as virtual services accessed in ways similar to other system services or utilities [13].

The remainder of the paper is organized as follows. Section 2 discusses the design of virtualized services, followed by their management in Section 3. Section 4 describes device enhancements made possible by virtualized services, which is followed by a description of the implementation and experimental evaluation in Section 5. Section 6 discusses related research, while Section 7 outlines future work and concludes the paper.

## 2 Design of Virtualized Services

Services afford the clean separation of an implementation from its use, and are a popular option for separating levels of abstraction in software. They may exist within a single machine, such as system calls, or across machines, like remote procedure calls. They may also differ in their interaction types (request-response, publish-subscribe, etc.) or their interfaces (object-based, web-based, etc.). This generality makes them widely applicable, leading to their extensive use and deployment in settings ranging from single operating systems [13] to distributed, service-based architectures [14].

Device virtualization in current systems uses a hosted virtual machine architecture [15], where only a single domain, termed the 'host', is permitted direct access to the physical devices (using the corresponding device drivers belonging to the OS running in the domain). I/O actions from other domains are directed by the hypervisor to this host, which mediates all accesses to the device. Paravirtualization techniques use a similar mechanism – for instance, Xen uses a split driver model where the device I/O of a guest domain is handled by the frontend device driver within the domain, which interacts with a backend in the control domain (i.e., the host) via the hypervisor's sharing infrastructure. The backend then uses native device drivers to directly access the hardware.

Technologies provided by the virtualization infrastructure to support efficient I/O via 'host' domains include (1) the provision of shared memory between VMs, (2) low latency message passing, particularly when guest VMs and host execute on different cores of a multicore platform, and (3) the use of parallelism and concurrency through effective use of multicore resources [16]. In the Xen hypervisor, for instance, shared memory is used for data transfer, and messaging is used to emulate interrupts, accessed via an event channel interface [3].

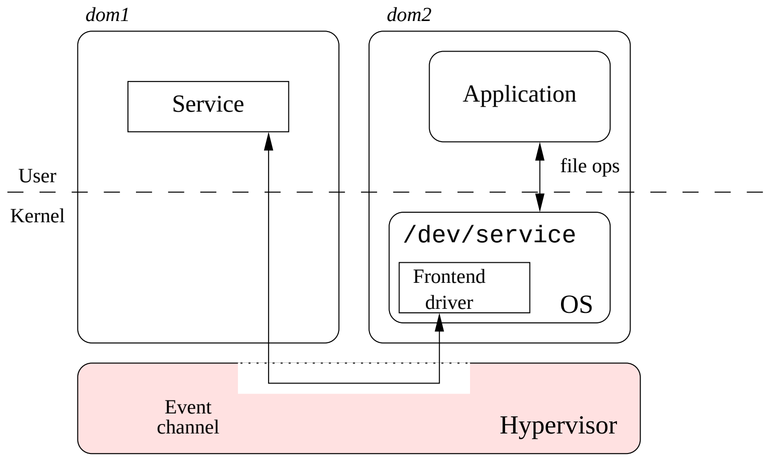


**Fig. 1.** Virtualized Services Architecture

### 2.1 Virtualized Services Architecture

Virtualized services – VServices – reuse the same mechanisms used by VMMs for device virtualization. Shared memory is employed to transfer data between the service provider and the consumer, and the messaging infrastructure is used for control (such as initialization, parameter changes, shutdown, etc.). Although these mechanisms do not impose any restrictions on the service interfaces and hence allow general services to be implemented and in keeping with other standards [13], we also reuse the file-based interface commonly used in all UNIX-based systems for the following reasons: (i) this interface is standardized so that adapting it to non-UNIX platforms is also straightforward, (ii) it makes bundling services with existing devices simpler, and (iii) since operating systems have been optimized around this interface, its efficiency does not pose a concern.

The overall architecture of VServices, as implemented in the Xen VMM hosting two Linux domains, is shown in Figure 1. The domain *dom1* acts as a service provider, and an application in domain *dom2* uses this service. The OS inside *dom2* provides a device frontend that exports a `/dev/service` file as a character device (termed service device). Any application requiring the service will access the service device via regular file operations. These are translated to the appropriate commands by the frontend device driver for the service, which in turn interacts with *dom1* via the event channel interface provided by the hypervisor.

In the remainder of the section, we discuss the design and implementation of two types of services: (1) directory and (2) group communication services, both realized using the VServices architecture.

### 2.2 Directory Service

A directory service uses a request-response based interaction to perform directory lookups based on request parameters. Common directory services include the

Domain Name Service (DNS), and LDAP-based address book or yellow pages lookups. A virtualized directory service implementation propagates writes to the service device into the domain that hosts the service (or acts as a proxy to the service). Here, the write is translated to a request call. The response returned by the service is communicated to the requesting domain via a soft IRQ. Upon a read, this response is passed on to the application. Additionally, ioctl() calls exported by the service device are used to set and get parameters (server name, buffer sizes, etc.). Errors from the service are transparently reported as appropriate file access errors. Due to the generic nature of the interface, this service can be extended to work as an RPC service (Sun-RPC, XML-RPC) or as a web service (SOAP-based) with small modifications.

### 2.3 Group Communication Service

The group communication service is realized using an event-based publish-subscribe middleware. Pub-sub middleware provides transparent construction and maintenance of multicast trees among participating nodes in a group, and exposes simple interfaces to enable communications (via publish and subscribe). In the virtualized group communication service implementation, writes from the application are translated to publish calls, and received events are passed to the application when read calls are issued. As in the directory service, ioctl() calls are used to set and get parameters (buffer sizes, for example) as well as to initiate control actions (creation of a channel, subscribing to a channel, etc.). Table 1 summarizes the interfaces used by these services.

**Table 1.** File operations and service semantics

| File operations | Directory | Pub-Sub |
|---|---|---|
| `open` | init | init |
| `close` | cleanup | cleanup |
| `write` | send request | publish event |
| `read` | receive response | receive event |
| `select/poll` | wait for response | wait for event |
| `ioctl` | control operations | channel management |

VServices are accessed by applications via the service device, and hence require applications to use the device interface instead of service calls. In order to allow legacy applications to use this interface, a library is used to provide wrapper calls that translate service calls to device accesses (Figure 2). Applications using the service via dynamic libraries require no changes to use VServices. However, those that are built via static linking need to be rebuilt to link with the wrapper library.

The entire state associated with each VService is stored in the guest VM's frontend (in addition to in the backend), and hence requires no changes to the

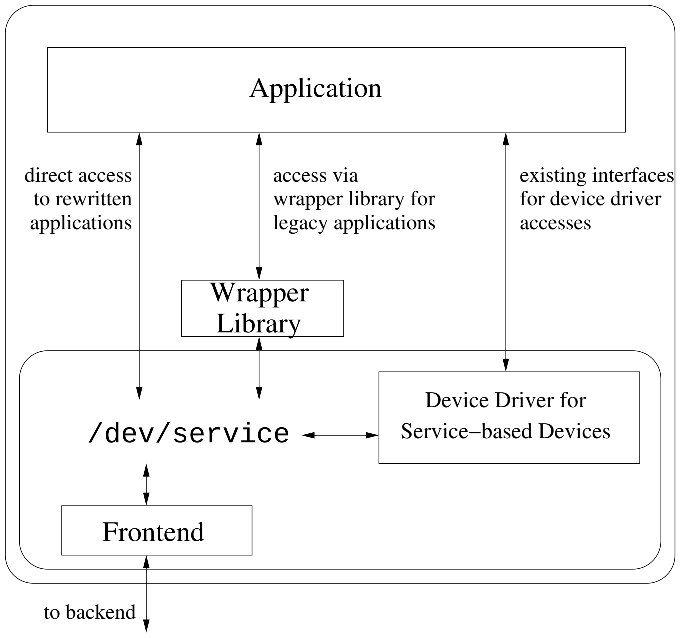

Application
direct access to rewritten applications
access via wrapper library for legacy applications
existing interfaces for device driver accesses
Wrapper Library
/dev/service
Device Driver for Service–based Devices
Frontend
to backend

**Fig. 2.** VService - Application Interactions

implementation of VM migration. On the commencement of migration, the backend simply discards its copy of the state corresponding to the VM, and the state is communicated to the backend on completion of a migration, which is then handled by the new backend. For instance, the channel identifier of a pub-sub channel is stored in the frontend as well as in the backend, and on migration, the new backend acquires the ID from the frontend and completes subscription, while the old backend unsubscribes from the channel and discards the channel ID.

VServices have been implemented and demonstrated using network-based services, but that does not constitute an inherent limitation. For instance, other capabilities such as vector processing, cryptographic operations, etc. can be offered as local services via the VService interface.

## 3 Virtualized Services Management

One of the principal advantages of using VSservices 'behind the scenes' is the opportunity to optimize their operation using modern middleware technologies. This typically involves active service management concerning service quality and/or performance metrics, as discussed next.

## 3.1 Performance Optimizations

Capabilities enabled by the centralized service implementations in VServices include the following.

*Concurrency:* As many service implementations within individual domains are avoided and the corresponding service access stack eliminated (for instance, a domain running a SOAP client application can get rid of its own implementations of HTTP and the entire network stack), this leads to lightweight OSes in guest domains with smaller memory footprints. Such consolidation of service implementations can also lead to more efficient service scaling (evaluated in Section 5.1), particularly when combined with appropriate shared state management. Additionally, any specialized processors or other hardware in the platform that can speed up service implementations can be utilized effectively, especially if a guest OS does not itself possess the capabilities to handle such hardware (e.g., vector co-processors).

*Sharing:* By sharing semantic information about their use, VServices can expose opportunities to share functionality. For instance, consider a virtualized DNS service where DNS requests and responses from guest domains are relayed to the nameserver, by the backend at the control domain. If the backend now caches the responses for the various requests locally, responses to previously unrelated requests from separate domains can possibly be sped up. On the other hand, sharing also opens up potential security/privacy concerns. In the same example, a domain receiving an immediate response to the first DNS request it makes could infer that some other domain had recently made the same request.

## 3.2 Quality Management

A key benefit of the fact that VServices are virtualized at levels of abstraction higher than those of low level device drivers – application virtualization – is the ability to associate with VService actions meaningful Service Level Agreements (SLAs). For instance, instead of guaranteeing network packet level metrics agnostic of the data movement, content-based service level guarantees can be provided, without the need to change or update communication protocols [17]. Further, these can be enhanced with policies for sharing services among domains, to address security related issues, as discussed in Section 3.1. Finally, the ability to decouple the quality of service implementation for a domain from its scheduling priority is also enabled, giving greater flexibility.

## 3.3 Limitations

VServices provide benefits as outlined in Sections 3.1 and 3.2, besides enabling the innovative functionality discussed in Section 4. On the other hand, VServices virtualize at levels of abstraction – at the application level – for which there may not exist well-defined, standard APIs. This can be remedied by emulating the standard file- or object-based interfaces used in operating systems or by exploiting subsystem-specific standards (e.g., using the Linux v4L interface [18] or the T10 standard for object-based file systems [5]. Another remedy is to use wrapper libraries OS virtualization, of providing an abstraction that is closest to the bare hardware [19]. An issue with all such solutions, however, is that during VM migration, additional support is required for dealing with the VServices being used. Since standard hypervisors do not provide such support, this results in the need for extensions in the 'host' domain.

## 4 Using Middleware for Device Enhancement

As stated earlier, a principal advantage of VServices, in part caused by its use of the standard split driver model used by all other devices, is that this provides us with the opportunity to transparently provision these services with new functionality or capabilities. Examples include extending device functionality, device emulation (discussed in Sections 4.1 and 4.2), device remoting (linking a local device driver with a remote device), device consolidation (using the functionalities of different devices to present a single emulated device), device functionality isolation, etc. In general, this is useful for providing a uniform set of functionalities to guest VMs, independent of the physical device's capabilities or the environment in which it is being used. Any deficiencies in the capabilities of the actual device are compensated by using services that emulate these features. We next describe several innovative device extensions implemented with VServices, which exploit middleware methods and realizations.

### 4.1 Extending Device Functionality

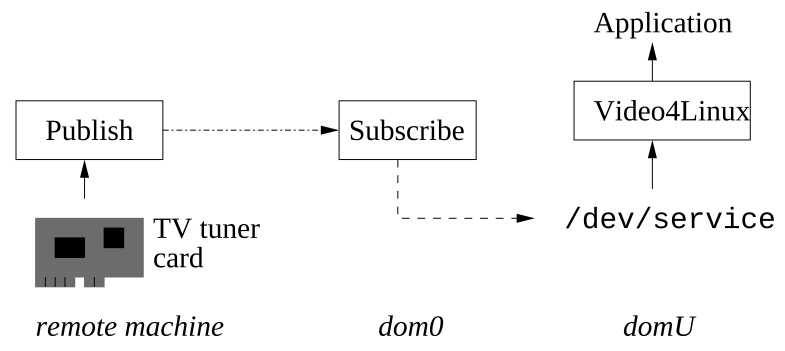


**Fig. 3.** VService-based TV tuner

Consider the common Video4Linux(v4l) device driver API to access multimedia devices such as camera, TV/radio tuners, encoder/decoders, etc. This multi-purpose interface can be used to provide a rich set of functionality to applications, invisibly provided by actual devices and/or provided by middleware realizations of these devices. An example is a VService that provides TV/radio tuner functionality to a guest VM although the physical device does not support it, by devising a middleware-realized service that provides the multimedia streaming content for the tuner device. This is one of the examples realized in this paper, where we use the capabilities of a USB-based webcamera to provide a v4l interface to the guest VMs, but enhance it to provide TV tuner capabilities as well. This is done by having the service backend subscribe to a channel in a publish-subscribe middleware for the media content. Figure 3 shows the design of such a device. Further, a mechanism like this can be extended to provide capabilities that do not exist in *any* physical devices at all. For instance, content from a location-based information service can be overlaid on content from the camera to provide location-based images. We realize such functionality by using a location- and orientation-based service that provides information such as the directions to nearest conveniences overlaid on the camera image from the webcam. An further extension of this notion allows users to compose a chain of services coupled with devices to provide rich platforms to guest VMs, resembling the system architecture for pervasive computing project described in [20, 21], or using simple techniques similar to web service mashups [22], already used in the OS context [23].

## 4.2 Device Emulation

The split driver model designed in virtualization infrastructures like Xen already supports simple implementations of emulated devices. While existing methods use specialized software to realize such emulations [18, 2], the novel contribution of this paper is to demonstrate how service-based middleware implementations can offer several advantages over such work, including gaining the flexibility to dynamically switch between different service implementations, for instance. The concrete example shown in this paper is a Global Positioning System (GPS) device, which in its standard implementation, provides location information based on timing differences of signals received from various GPS satellites, but in a second implementation, is emulated indoors using two different service implementations: (i) using the strength of signals from various bluetooth devices in the vicinity and the predetermined knowledge of these signals at various locations in the area [12], and (ii) detecting the position using external cameras that detect and track the target, then deduce its position based on a precalibrated scale. Each of these are available as services, and the backend can choose the service to be used based on factors such as the environment (i.e., whether the target is in the camera's range), accuracy desired (between the two methods), power consumed (bluetooth vs. camera), etc.

Figure 4 shows the design of such a VService-based device, where the shaded boxes represent remote components and the clear boxes, local ones. Here, we assume that although the bluetooth signal strengths are determined by the local device, an external service that translates signal strengths to precise coordinates (based on previous surveys) exists as a network-based service. Current GPS devices are typically accessed via the serial port, and vary slightly in their interface to applications. As a result, higher level daemons have been developed (e.g., gpsd in Linux) to present a standard interface to applications. VService-based localization can be emulated as a serial device, or allow applications to access them via a modified gpsd wrapper daemon.

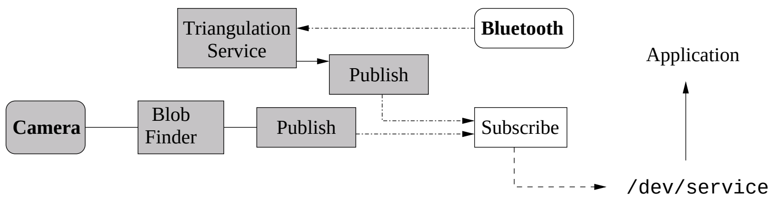


**Fig. 4.** VService-based indoor localization

## 5 Experimental Evaluation

VServices are implemented in Xen 3.0.4 running a Linux kernel 2.6.16.33. Experiments are conducted on a Dell Latitude notebook with a 1.66GHz Intel Centrino Duo dual-core processor, and 1GB memory. The control domain, or dom0, uses both cores for execution. All guest VMs share one of the two cores, with each VM allocated 64MB of memory. A directory service and a publish-subscribe service are implemented. The directory service is implemented using the domain name service as an example. It translates `write()`'s of server names to IP addresses, which are then accessed using `read()` calls. The publish-subscribe service uses the `ioctl()` call for channel management and performs buffer sizing, as discussed in Section 2.3. The backend implements the ECho pub-sub middleware [24], a portable event-based middleware which provides support for installing event filters at run-time using dynamically generated code. However, the interface is general enough to allow substitution of other event-based pub-sub middleware in its place. Time measurements use the `rdtsc` instruction for accuracy.

### 5.1 Service Costs and Scalability

Our first set of experiments measure the costs involved in performing DNS service calls, with the hostname of another host in the same LAN, as the request. As Table 2 shows, the main contribution is from the `write()` call, which signals the backend about the new request and blocks on the response. The backend directs the request to one of the threads in a thread pool, which performs a `gethostbyname()` call. This involves searching the local cache (`/etc/hosts` in Linux) for a match and if found to be missing, obtaining the IP from the nameserver. The `init()` and `connect()` calls are involved in opening and connecting a socket to the nameserver, while `send()` and `recvfrom()` perform the lookup, following which the socket is closed. The blocking `recvfrom()` call takes up about 90% of the overall backend costs. `read()` performed at the guest VM is a simple copy from the buffer. `close()` releases the thread back to the pool.

**Table 2.** Cost breakdown for the DNS VService, star indicates a backend operation

| Operation | Cost (cycles) |
|---|---:|
| `open` | 4232 |
| `write` | 736691 |
| $*$ socket `init` | 30760 |
| $*$ `connect` | 5660 |
| $*$ `send` | 14247 |
| $*$ `recvfrom` | 564064 |
| $*$ socket `close` | 11154 |
| `read` | 7175 |
| `close` | 35543 |

Figure 5 compares the overall cost of performing the DNS call using the VService implementation, against the traditional `gethostbyname()` implementation from the guest VM. `gethostbyname()` takes roughly 1ms for each lookup, when only one VM is present, and shows a small increase for up to 4 VMs, but shoots up rapidly to over 7ms for 8 VMs. This is because of duplicated work in guest VMs. Specifically, since each VM performs these operations over its own TCP stack and relies on the control domain only for access to the physical device. In contrast, the VService implementation scales better since most of the work is performed by the control domain. Even for a single domain, VService offers about 50% lower latency. The benefits are entirely due to low overheads, since no caching of IP is undertaken in the control domain (i.e., each DNS request is propagated to the nameserver in all the cases).

Next, we evaluate VServices performance, using the pub-sub implementation. In these experiments, another host in the same network creates a channel, and applications inside the VMs publish or subscribe to the channel. During each run, 1000 events of size 4KB are sent continuously by the publisher(s), and several runs are conducted. Figure 6 shows the performance of publish and subscribe operations, as the number of VMs (and hence the number of publishers/ subscribers) is increased. It compares the naive implementation that uses the guest VM's virtual network, against the use of VServices. In the case of publish, we notice that the naive implementation performs better than VService, but scales worse. The primary reason for this is that publish operations translate to blocking `write()` calls on the device, which are handled only in batches by the backend (in order to minimize frequent VM context switches). This can be rectified by having a thread continuously handle these operations at the backend, without waiting for explicit signals from the frontend. While this may improve performance, it can also increase CPU utilization. Alternatively, by changing the buffer size, it is possible to obtain better throughput (discussed in Section 5.2).

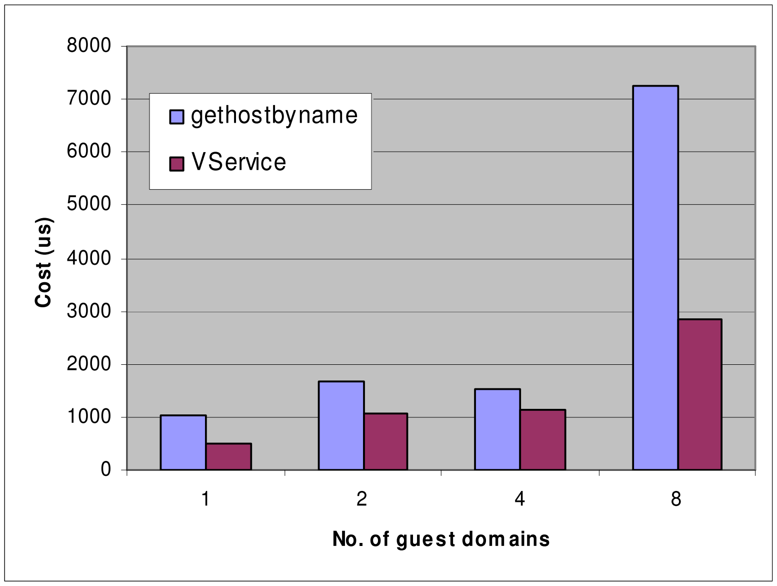
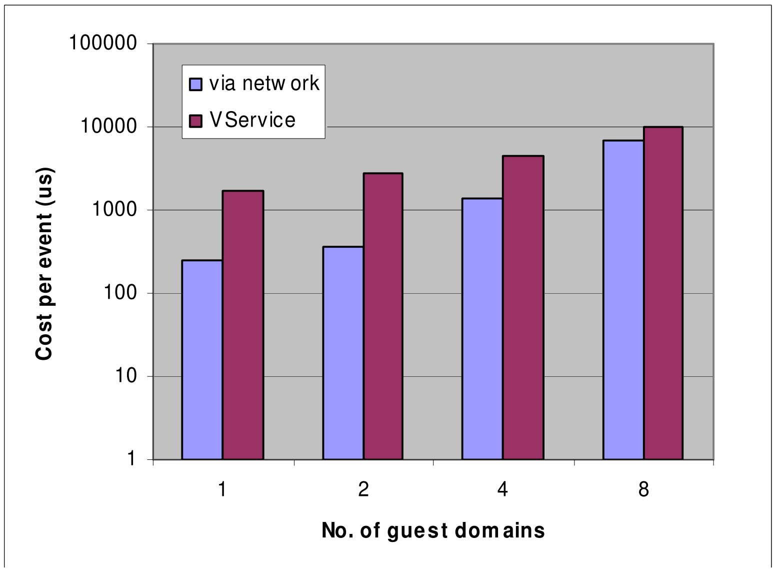


**Fig. 5.** DNS Costs Comparison and Scalability

In the subscribe scenario, we find almost equivalent performance with the naive- and VService-based subscribes for the single VM case, but the latter again scales much better than the former – in fact, it almost stays constant, as the number of VMs increases. This behavior stems from two factors: (i) the implementation of the ECho [24] middleware we are using in this VService implementation, and (ii) the VService `read()` implementation. Firstly, since each subscriber in the naive implementation has a unique IP address, a multicast tree is constructed by ECho such that each VM becomes an endpoint. The same events traverse through multiple virtual interfaces to reach the endpoints. In the case of VService, there exists only one such endpoint, and the events are copied to the buffers corresponding to each VM, which are then directly read by the applications. Since these copies are inexpensive, they do not significantly add to the overall cost. Secondly, subscribe buffers are implemented as ring buffers, where the application can read from the buffer concurrently with backend's `write`s to it. As a result, the throughput is not affected as long as the CPU is not fully-utilized.

## 5.2 Services Management

With the use of multiple, or specialized cores, allowing the backend to efficiently manage the core assignment to various operations becomes possible, as discussed in Section 3. In this experiment, we evaluate the assignment of the VService backend operations to the same core as the VM, or to the other core, and report



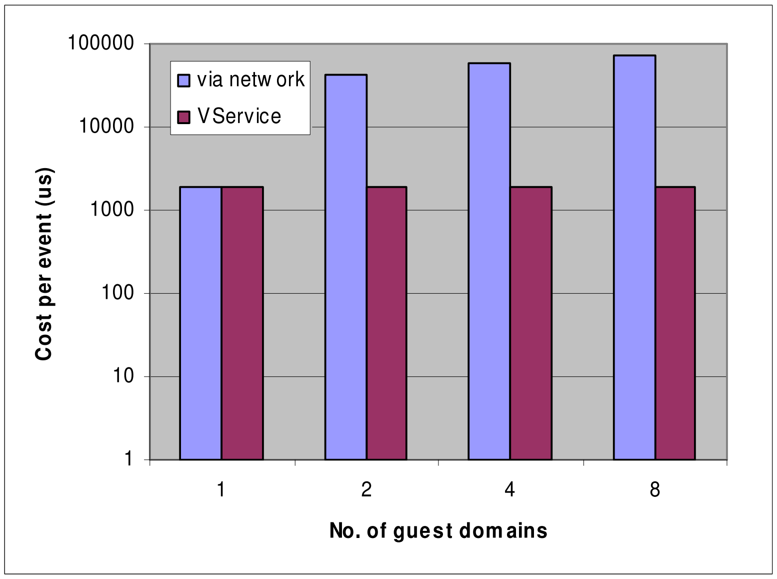


**Fig. 6.** Costs Comparison and Scalability for Publish and Subscribe Operations

our findings in Table 3. While the DNS example has only one running VM, the pub-sub example has two VMs (no gains were found with only one VM). However, the gains are found to be very small (less than 5% in all the cases). We may attribute this to the fact that since VService is implemented using blocking `write()`'s, the service consumer is idle during service fulfillment as well, and this dependency limits parallelism. In the case of subscribe, we find that the backend is able to fill the subscribe buffers with incoming events much faster than the frontend can read and pass them on to the application. This stems from the batched nature of event handling (in order to minimize VM context switches), that allows for limited gains. We conclude that, to realize the potential of multiple cores, a redesign of the current VServices implementation to allow for non-blocked writes and unbatched reads is necessary.

**Table 3.** Effect of core use

| VService | Cost (us) | |
|---|---|---|
| | same core | different core |
| DNS (1000 requests) | 412.4 | 407.1 |
| Publish (per event) | 3013.4 | 2898.4 |
| Subscribe (per event) | 1902.0 | 1902.6 |

VServices are implemented with a total buffer size of 64KB, but this can be changed easily. If read and write buffer sizes are changed to smaller (larger) values, this implies that the frontend will signal the backend more (less) frequently. As `write()`s are currently implemented as blocking writes, and `read()`'s buffers are implemented as ring buffers, this leads to differences in their behavior with respect to buffer size changes. Since each event is 4KB in size, writes would block for each event when the buffer size is 4KB or less, whereas with a 32KB buffer, every eighth write is blocked before it is handled, along with the previous seven writes. This allows latency to be traded off for throughput. `read()`s do not follow this behavior and show steady performance, for reasons previously mentioned in Section 5.1. Figure 7 shows the results, which closely follow our discussion. Note that these effects do not apply to the DNS service, since latency is important in DNS interactions. Consequently, for that VService, *any* write/read is handled immediately by the other end.

## 5.3 Device Enhancements

To implement the TV tuner VService example, we use another Linux desktop in the same network with a 3GHz dual-core Pentium 4 processor and 512MB memory, that possesses a Conexant Brooktree 878 PCI-based TV tuner card to act as the media server, connected through a 100Mbps switch. We use the fftv-0.8.3 open source software to grab frames from the card using the V4L driver, and publish it using ECho (Figure 3). VService is used to subscribe to this channel and to obtain the frames inside the guest VM. The three different strategies we evaluate in this experiment are: (i) Local frame grab using the `read()` system call, (ii) Local frame grab using the `mmap()` system call, and (iii) Remote frame grab using VService, that uses the `mmap()` based call to access the TV tuner. Frames are continuously grabbed and the period between frames noted. We also evaluate three sizes of images – 160x128 pixels (large), 96x64 (medium), and 48x32 (small). These sizes are chosen in order to avoid the need for compression.

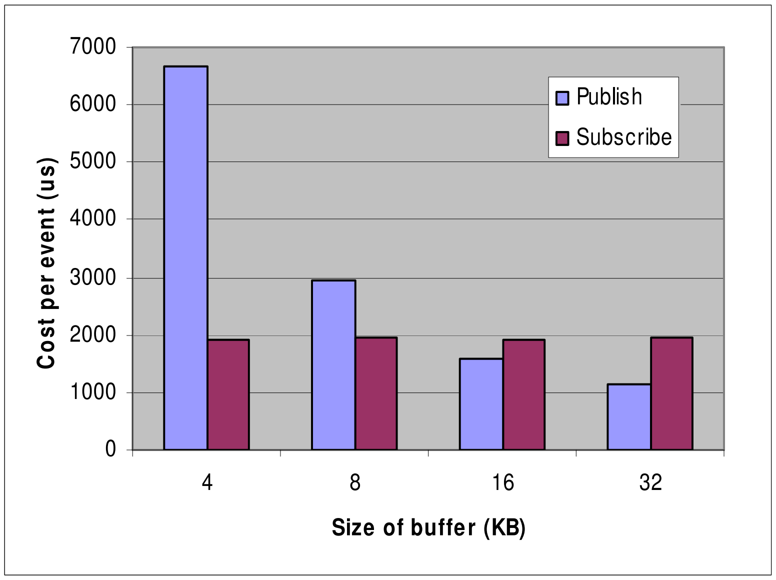


**Fig. 7.** Effects of VService buffer sizes on pub-sub costs

The same software (fftv-0.8.3) is used within guest VMs to obtain the frames from the driver to the application.

Experimental results are shown in Figure 8, along with the jitter values represented using error bars. VService adds a 50% overhead to `mmap()` based calls while transferring the frames across the network, and higher jitter values to smaller frames. It is noted that these costs are comparable to local `read()` based frame grabs.

For the indoor localization device, we present costs for each of the operations involved in obtaining the data. We use a Logitech Quickcam USB-based webcam with the gspca-based V4L driver to capture images, then process them using the CMVision blobfinder tool, which detects objects with pre-defined colors in the image as "blobs". The position of the blob is then translated to absolute physical coordinates, using precalibrated readings (the camera is assumed to be static). These coordinates are published through an ECho channel. The bluetooth-based localization is performed using a USB-based Belkin bluetooth adapter attached to the local machine. We assume the presence of other fixed bluetooth devices in the vicinity that act as beacons, as mentioned in Section 4.2. We present the basic costs of this service in Table 4. The time taken to grab a frame, process it, and receive the coordinates add up to less than 5ms. The time taken to determine link quality and infer coordinates might be slightly higher (depending on the number of beacons, as well as link quality to coordinate conversions), but this can still be performed well under 1s, which is the latency typically offered by outdoor GPS devices. Despite the low latency in determining the link quality of a bluetooth device, the time involved to scan for other devices and connect

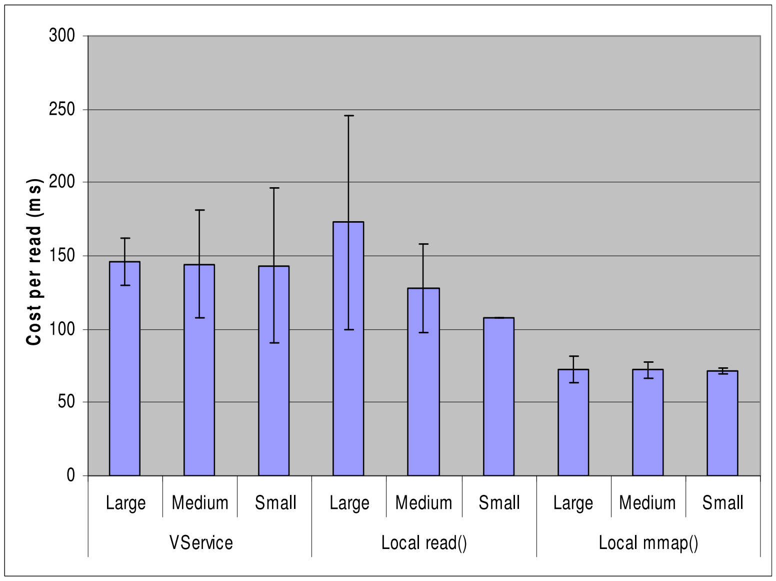


**Fig. 8.** Cost per frame using different grab techniques, on different frame sizes

to them are significantly higher. Fortunately, these are one-time operations, and do not introduce additional latencies once a connection has been established.

**Table 4.** GPS device component costs

| Component | Cost (us) |
|---|---|
| Frame grab | 108.33 |
| Blob detection | 4210.18 |
| Bluetooth scan (s) | 11.85 |
| Bluetooth connect (s) | 6.47 |
| Bluetooth link quality | 5095.40 |
| Pub-sub delay | 213.25 |

## 6 Related Work

Recent efforts in middleware have resulted in implementations based on the concepts of Service Oriented Architecture (SOA) [14], and service virtualization [25]. SOA eases the creation of enterprise applications by allowing the composition of various services that serve to accomplish specific tasks, to align with business processes. Service virtualization helps in adapting the SOA model to a heterogeneous environment, by providing consistent interfaces for its management. Combining middleware concepts with virtualization mechanisms, it benefits from the well-defined interfaces provided by SOA and from management mechanisms such as load balancing, migration, isolation and resource monitoring provided by virtualization. As a result, it is gaining adoption in grid computing [26]. VService combines services with virtualization techniques at a lower level, thus making device emulation and composition possible, by interfacing services with device drivers. Service Oriented Device Architecture (SODA) [27] allows accesses to devices via services (in contrast to the VService approach of presenting services via device interface), in order to simplify device management. It is developed for use in enterprise systems, where the same interfaces designed to access enterprise services are also used to access and control devices (such as RFID tags, and other sensors or actuators). UPnP [28] (Universal Plug and Play) is a middleware standard designed to allow linking of devices in a seamless manner. Targeting personal area networks such as a home network, it aims at minimizing user involvement in setup. VService can be easily adapted to such a setup, by replacing the directory or group communication services demonstrated in this paper with UPnP compatible protocols. The use of component middleware to implement resource-intensive applications such as software defined radios, radar systems, etc., using parallelization techniques is described in [29].

Use of the device interface for implementing services is a well-established practice, originating with Unix file-based APIs. Examples include the use of the `/dev/random` device in several flavors of UNIX to generate pseudo-random numbers in software. Other examples include the `/dev/evtchn` in Xen [3], [30] to exchange events between domains, and `/dev/binder` in Openbinder [31] to exchange data between components. The Plan 9 operating system [13] extended this idea to include all resources in the system to be available via the file system interface. For instance, a TCP connection is made by accessing files under `/net/tcp`.

The Libra [11] library operating system extends the Exokernel idea [32] of providing customized operating system to applications, thus delivering only the functionality needed by the applications. Libra provides services required by a Java application running within a JVM, by implementing frequently accessed services locally and relying on the hypervisor for other services. Libra uses the 9P distributed file system protocol to access remote services, whereas VService offers common services implemented by the control domain, for use by other VMs. Both these efforts are valuable in minimizing the size of guest operating systems.

There are multiple ways to virtualize devices. Xen and VMware use the methods outlined in Section 2. KVM [33] uses similar methods, but relies on Qemu [34] running on the host kernel for device emulation. User mode Linux [35] relies on the parent kernel for device access, since its kernel runs as a user-space process. In order to improve device virtualization, providing exclusive unmediated control of a device by a single VM has been studied. The PCI passthrough [36] feature in Xen allows such access to the PCI device by a single VM, and thus avoids control domain overheads, but it also precludes sharing. Efforts to improve the network device virtualization in Xen VMM include Xensocket [37], which recommends bypassing the TCP stack and using copy instead of page flipping, when exchanging data between domains. This shows significant improvements in throughput, especially with large message sizes. VServices similarly avoids the guest VM TCP stack for network accesses, allowing the backend to package the data on the guests' behalf. Other research has proposed several network optimizations to Xen, which include performing optimizations either with hardware (if supported) or in the control domain (if hardware doesn't support these optimizations) [38]. The VService approach allows such an optimization to be cleanly implemented for services by completely bypassing the virtual network interface. Vmedia [18] and Netchannel [2] are examples of efforts aimed at extending device functionality and transparent use of remote devices in the VM context, respectively.

## 7 Conclusions and Future Work

This paper demonstrates the novel use of middleware in conjunction with virtualization techniques to provide performance benefits and new functionalities to virtual machines. It introduces the notion of VServices, using which existing virtual device interfaces in virtualization systems – currently used to share physical devices among virtual machines – are extended to cleanly share arbitrary services. Improvements in performance, via lower latency, higher throughput and better scalability are demonstrated using two example services implemented via the Xen VMM. The usefulness of VService via device enhancements is also shown, by adapting it to use a remote TV tuner card, and with an indoor localization service.

VServices currently use custom protocols to implement the various services discussed. The 9P distributed filesystem protocol is a standard protocol used in several different efforts to implement remote services, and our future work may attempt to integrate 9P with VService. This paper is part of a larger effort [18], [2], [5] that explores performing virtualization at higher layers, thus enabling the management and sharing of resources by backends, depending on usage semantics. Other research in this space include VMGL [39] which performs virtualization at the OpenGL interface layer, and the Boxwood project [40] of the Singularity operating system that provides storage at higher abstraction levels such as B-Trees for example, hiding lower level disk virtualization details from applications. As noted before, VServices depart from the basic virtualization philosophy of providing maximal control to operating systems in guest domains. One reason for doing so is to explore the opportunities of concurrency and sharing in future multicore platforms, including those targeting the mobile and pervasive domain. Our future efforts will go beyond the basic VServices implementations shown in this paper to further explore their use with handheld devices and mobile applications, in joint work with collaborators at Motorola Labs.

## References

1. Nelson, M., Lim, B.H., Hutchins, G.: Fast transparent migration for virtual machines. In: USENIX Annual Technical Conference. (2005)
2. Kumar, S., Schwan, K.: Netchannel: a vmm-level mechanism for continuous, transparentdevice access during vm migration. In: ACM VEE. (2008)
3. Pratt, I., et al.: Xen 3.0 and the Art of Virtualization. In: Proc. of the Ottawa Linux Symposium. (2005)
4. : The VMWare ESX Server. `http://www.vmware.com/products/esx/`
5. Raj, H., Schwan, K.: O2s2: Enhanced object-based virtualized storage. In: SPEED. (2008)
6. Jondral, F.K.: Software-defined radio: basics and evolution to cognitive radio. EURASIP J. Wirel. Commun. Netw. **5**(3) (2005) 275–283
7. Davies, N., Wade, S.P., Friday, A., Blair, G.S.: Limbo: a tuple space based platform for adaptive mobile applications. In: ICODP/ICDP. (1997)
8. Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R.E., Sturman, D.C.: An efficient multicast protocol for content-based publish-subscribe systems. In: ICDCS. (1999)
9. Mascolo, C., Capra, L., Emmerich, W.: Mobile computing middleware. Advanced Lectures on Networking **2497** (2002)
10. Gaddah, A., Kunz, T.: A survey of middleware paradigms for mobile computing. Technical Report SCE-03-16, Carleton University (2003)
11. Ammons, G., Appavoo, J., Butrico, M.A., Silva, D.D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B.S., Hensbergen, E.V., Wisniewski, R.W.: Libra: a library operating system for a jvm in a virtualized execution environment. In: Proc. of VEE. (2007)
12. Hossain, M., Soh, W.S.: A comprehensive study of bluetooth signal parameters for localization. In: IEEE PIMRC. (2007)
13. Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., Winterbottom, P.: Plan 9 from Bell Labs. Computing Systems **8**(3) (1995)
14. Papazoglou, M., Georgakopoulos, D.: Service-oriented computing. Communications of the ACM **46**(10) (2003)
15. Sugerman, J., Venkitachalam, G., Lim, B.H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In: Proc. of USENIX ATC. (2001)
16. Kumar, S., Raj, H., Ganev, I., Schwan, K.: Re-architecting vmms for multicore systems: The sidecore approach. In: Workshop on the Interaction between Operating Systems and Computer Architecture. (2007)
17. He, Q., Schwan, K.: Iq-rudp: Coordinating application adaptation with network transport. In: HPDC. (2002)
18. Raj, H., Seshasayee, B., Schwan, K.: Vmedia: Enhanced multimedia services in virtualized systems. In: Proceedings of Multimedia Computing and Networking. (2008)
19. VMware white paper: Understanding Full Virtualization, Paravirtualization and Hardware Assist. `http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`
20. Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., Wetherall, D.: System support for pervasive applications. ACM Trans. on computer systems **22**(4) (2004)
21. Roman, M., Ziebart, B., Campbell, R.H.: Dynamic application composition: Customizing the behavior of an active space. In: PERCOM. (2003)

22. Jackson, C., Wang, H.: Subspace: secure cross-domain communication for web mashups. In: Proc. of Intl. conf. on WWW. (2007)
23. Howell, J., Jackson, C., Wang, H., Fan, X.: Mashupos: Operating system abstractions for client mashups. In: Proc. of USENIX HotOS. (2007)
24. Eisenhauer, G., Bustamante, F.E., Schwan, K.: Event services in high performance systems. Cluster Computing: The Journal of Networks, Software Tools, and Applications **4**(3) (2001)
25. Xu, M., Hu, Z., Long, W., Liu, W.: Service virtualization: Infrastructure and applications. The Grid: Blueprint for a New Computing Infrastructure (2004)
26. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: "grid services for distributed system integration". Computer **35**(6) (2002)
27. de Deugd, S., Carroll, R., Kelly, K.E., Millett, B., Ricker, J.: Service oriented device architecture. IEEE Pervasive Computing **5**(3) (2006)
28. UPnP: Universal Plug and Play forum. `http://www.upnp.org/`
29. Schmidt, D.C., Gokhale, A., Gill, C.D.: Patterns and performance of real-time and data parallel corba for high-performance embedded computing applications. In: HPEC. (2002)
30. XenSource: Xenintro. `http://wiki.xensource.com/xenwiki/XenIntro`
31. Hackborn, D., et al.: Openbinder. `http://www.open-binder.org/`
32. Engler, D.R., Kaashoek, M.F., O'Toole, J.: Exokernel: An operating system architecture for application-level resource management. In: Symposium on Operating Systems Principles. (1995) 251–266
33. Kivity, A. and Kamay, Y. and Laor, D. and Lublin, U. and Liguori, A.: kvm: the linux virtual machine monitor. In: Ottawa Linux Symposium. (2007)
34. Fabrice Bellard: Qemu, a fast and portable dynamic translator. In: Proc. of USENIX ATC. (2005)
35. Jeff Dike: User-mode linux. In: Proc. Linux showcase and conference. (2001)
36. Zana, G.: Hvm pci passthrough. XenSummit (2007)
37. Zhang, X., McIntosh, S., Rohatgi, P., Griffin, J.L.: Xensocket: A high-throughput interdomain transport for virtual machines. In: Proc. of Middleware. (2007)
38. Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in xen. In: Proc. of USENIX ATC. (2006)
39. Lagar-Cavilla, H., Tolia, N., Satyanarayanan, M., Lara, E.: Vmm-independent graphics acceleration. In: Proceedings of the 3rd international conference on Virtual execution environments. (2007) 33–43
40. MacCormick, J., Murphy, N., Najork, M., Thekkath, C., Zhou, L.: Boxwood: Abstractions as the foundation for storage infrastructure. In: OSDI. (2004)