VirtuOS: an operating system with kernel virtualization

Ruslan Nikolaev, Godmar Back
rnikola@vt.edu, gback@cs.vt.edu
Virginia Tech
Blacksburg, VA

Abstract

Most operating systems provide protection and isolation to user processes, but not to critical system components such as device drivers or other system code. Consequently, failures in these components often lead to system failures. VirtuOS is an operating system that exploits a new method of decomposition to protect against such failures. VirtuOS exploits virtualization to isolate and protect vertical slices of existing OS kernels in separate service domains. Each service domain represents a partition of an existing kernel, which implements a subset of that kernel's functionality. Unlike competing solutions that merely isolate device drivers, or cannot protect from malicious and vulnerable code, VirtuOS provides full protection of isolated system components. VirtuOS's user library dispatches system calls directly to service domains using an exceptionless system call model, avoiding the cost of a system call trap in many cases.

We have implemented a prototype based on the Linux kernel and Xen hypervisor. We demonstrate the viability of our approach by creating and evaluating a network and a storage service domain. Our prototype can survive the failure of individual service domains while outperforming alternative approaches such as isolated driver domains and even exceeding the performance of native Linux for some multithreaded workloads. Thus, VirtuOS may provide a suitable basis for kernel decomposition while retaining compatibility with existing applications and good performance.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the Owner/Author(s).
SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522719

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management; D.4.5 [**Operating Systems**]: Reliability; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance

General Terms: Design, Performance, Measurement, Reliability

Keywords: Operating systems, virtualization, microkernel, hypervisor, Xen, driver isolation, IOMMU, exceptionless system calls

1 Introduction

Reliability and fault resilience are among the most important characteristics of operating systems (OS). Modern general purpose operating systems require that an application runs in its own protected virtual address space. System critical data resides in the kernel's address space where it cannot be directly accessed by applications.

This isolation protects user processes from each other and the kernel from misbehaving user processes, but falls short of protecting the system from failing kernel components. Failure of just one kernel component generally causes the entire system to crash. Major offenders are device drivers [20, 23, 28], which reportedly caused 65-83% of all crashes in Windows XP [20,28,41]. These components are numerous, hardware specific, and often less tested due to a more limited user base. Common causes of such bugs include memory overruns, improper use of resources and protocols, interrupt handling errors, race conditions, and deadlocks [46].

Architectural approaches for increasing the reliability of kernel software and reducing the impact of faults often rely on *decomposition*. Microkernel-based system design moves device drivers and other system critical code from the kernel into separate user space processes. Microkernels have been successful in certain areas, but they require careful engineering to achieve good IPC performance [25, 36]. If application or driver compatibility with existing systems is required, either extensive emulation interface layers must be implemented [24],

or a Multiserver OS [22, 29] must be built on top of the microkernel, which partitions an existing monolithic kernel's functionality into multiple, independent servers that communicate with user processes via IPC.

Virtual machines also use hardware isolation to create strongly isolated domains in which to separate software components. Their design was driven by a need to safely share a machine's resources while maintaining application and kernel compatibility with existing systems. They, too, require careful consideration and optimization of their inter-VM and VM to hypervisor communication [26,38,47].

Despite their dissimilar motivations and development history, virtual machines and microkernels are connected and occupy the same larger design space [24,26]. For instance, microkernels were used as a foundation for virtual machine monitors [18], or even to support the reuse of existing device drivers [35] within the confines of a virtual machine. Conversely, modern hypervisors were used to achieve microkernel-like isolation of system components, such as in Xen's isolated driver domains [19].

This paper presents VirtuOS, a novel design that occupies a new point in this design space. VirtuOS uses hardware-based virtualization to encapsulate vertical slices of kernel functionality in isolated *service domains*. VirtuOS allows its user processes to interact directly with service domains through an exceptionless system call interface [48], which can avoid the cost of local system calls in many cases. Individual user processes may have system calls serviced by different service domains, which can provide redundancy when operating separate service domains for separate groups of user processes.

Each service domain handles a specific kernel service, such as providing a networking stack or a file system implementation, along with housing the drivers to access the underlying physical devices. A service domain runs a near-stock version of a kernel, including the major related components such as the socket layer, TCP/IP implementation, and (unchanged) device drivers. We use the PCI passthrough and IOMMU facilities of hardware-based virtual machine monitors to provide service domains with direct access to physical devices.

VirtuOS intercepts system calls using a custom version of the C library that dispatches system calls to the appropriate service domains. System call parameters and associated data are passed in shared memory that is accessible to the user processes and the service domain. To exploit the benefits of the exceptionless system call interface, the user library implements an M:N threading model whose scheduler cooperates directly with worker threads in the service domains.

We developed a prototype which implements a networking and a storage service domain to demonstrate the feasibility of this design. We used the Xen hypervisor along with the Linux kernel for VirtuOS's domains, along with a modified version of the uClibc [3], NPTL [16] and libaio [2] libraries. We tested our system with a wide range of server and client programs such as OpenSSH, mySQL, Apache, and Firefox.

We evaluated the performance for server-based workloads both under failure and non-failure scenarios. We found that VirtuOS can recover from the failure of individual service domains after restarting those domains and the processes that were using them. For network throughput tests and multithreaded transaction processing benchmarks, we found that VirtuOS meets or exceeds the performance of not only a split-driver model but also native Linux, indicating that it retains the performance benefits of exceptionless system call dispatch for those workloads. The performance loss for applications that do not benefit from the exceptionless model remains within a reasonable range.

The technical contributions of this paper are the following: (1) An approach to partitioning existing operating system kernels into service domains, each providing a subset of system calls; (2) A method for intercepting and demultiplexing of system calls using a user library and the dispatching of remote calls to service domains using an exceptionless mechanism; (3) A way to coordinate process and memory management in the primary and service domains so that applications can make transparent use of service domains.

We further discuss the challenges and limitations of our approach to partitioning as well as lessons we learned during the implementation of our prototype.

2 Background

This section discusses the system structure of the Xen hypervisor upon which VirtuOS relies, and it provides background regarding the exceptionless system call handling technique used.

2.1 Xen

Xen [8] is a widely used Type I hypervisor that allows the execution of virtual machines in guest domains. Originally developed for the IA32 architecture, which lacked secure virtualization capabilities [45], early versions of Xen required guest operating system code adaptations (i.e., paravirtualization [54]) to function. Processor vendors later introduced hardware virtualization extensions such as VT-x and AMD-V, which provide a VMM mode that is distinct from the mode in which privileged guest kernel code executes and which allows

the trapping or emulation of all sensitive instructions. Recent architectures add support for MMU virtualization via nested paging and support for the virtualization of memory-mapped I/O devices (IOMMU), which are supported by Xen's hardware containers (HVM). Xen's PVHVM mode, which we use for VirtuOS's service domains, adds additional paravirtualization (PV) facilities to allow guests to more efficiently communicate with the underlying hypervisor.

Xen's hypervisor implementation does not include device drivers, but multiple options exist to provide domains with access to devices. The original Xen design assigned device management to a dedicated, privileged domain (Dom0) using the device drivers present in that domain's guest kernel implementation. Other domains accessed devices using a split device driver architecture in which a front-end driver in a guest domain communicates with a back-end driver in Dom0. This design required the introduction of efficient interdomain communication facilities to achieve good performance. These include an interdomain memory sharing API accessed through guest kernel extensions, and an interrupt-based interdomain signaling facility called event channels. Split drivers use these facilities to implement I/O device ring buffers to exchange data across domains. The back-end drivers need not necessarily reside in Dom0 - the split driver model also provides the option of placing drivers in their own, dedicated driver domains [19, 47]. Though VirtuOS does not use a split driver model, it uses both the shared memory facilities and event channels provided by Xen to facilitate communication with service domains.

To achieve safe I/O virtualization, VirtuOS relies on two key facilities: PCI passthrough and the presence of an IOMMU. Xen's PCI passthrough mode allows guest domains other than Dom0 direct access to PCI devices, without requiring emulation or paravirtualization. These guests have full ownership of those devices and can access them using unchanged drivers. To make PCI passthrough safe, the physical presence of an IOMMU is required. An IOMMU remaps and restricts addresses and interrupts used by memory-mapped I/O devices. It thus protects from faulty devices that may make errant DMA accesses or inject unassigned interrupts. Thus, devices and drivers are isolated so that neither failing devices nor drivers can adversely affect other domains.

Self-virtualizing hardware [43] goes a step further by making a device's firmware aware of the presence of multiple domains and providing support for multiplexing its features to them. VirtuOS should be able to benefit from this technology as it emerges.

2.2 Exceptionless System Calls

Traditional system call implementations rely on an exception-based mechanism that transitions the processor from a less privileged user mode to kernel mode, then executes the system call code within the context of the current thread. This approach imposes substantial costs, both direct costs due to the cycles wasted during the mode switch, and indirect costs due to cache and TLB pollution caused by the different working sets of user and kernel code.

Exception-less system calls [48] avoid this overhead. Instead of executing system calls in the context of the current task, a user-level library places system call requests into a buffer that is shared with kernel worker threads that execute the system call on the task's behalf, without requiring a mode switch. Effective exceptionless system call handling assumes that kernel worker threads run on different cores from the user threads they serve, or else the required context switch and its associated cost would negate its benefits. A key challenge to realizing the potential gains of this model lies in how to synchronize user and kernel threads with each other. Since application code is generally designed to expect a synchronous return from the system call, user-level M:N threading is required, in which a thread can context-switch with low overhead to another thread while a system call is in progress. Alternatively, applications can be rewritten to exploit asynchronous communication, such as for event-driven servers [49]. VirtuOS uses the exceptionless model for its system call dispatch, but the kernel worker threads execute in a separate virtual machine.

3 Design & Architecture

VirtuOS's primary goal is to explore opportunities for improved isolation of kernel components in virtualized containers without significant compromises in performance. We present the architecture of our system in Figure 1. Our design partitions an existing kernel into multiple independent parts. Each part runs as a separate *service domain*, which represents a light-weight subset of kernel functionality dedicated to a particular function. A single, *primary domain* is dedicated to core system tasks such as process management, scheduling, user memory management, and IPC.

Service domains do not run any user processes other than for bootstrapping and to perform any necessary system management tasks related to a domain's function. Our design attempts to minimize the number of these processes, because the sole task of service domains is to handle requests coming from the user processes managed by the primary domain.

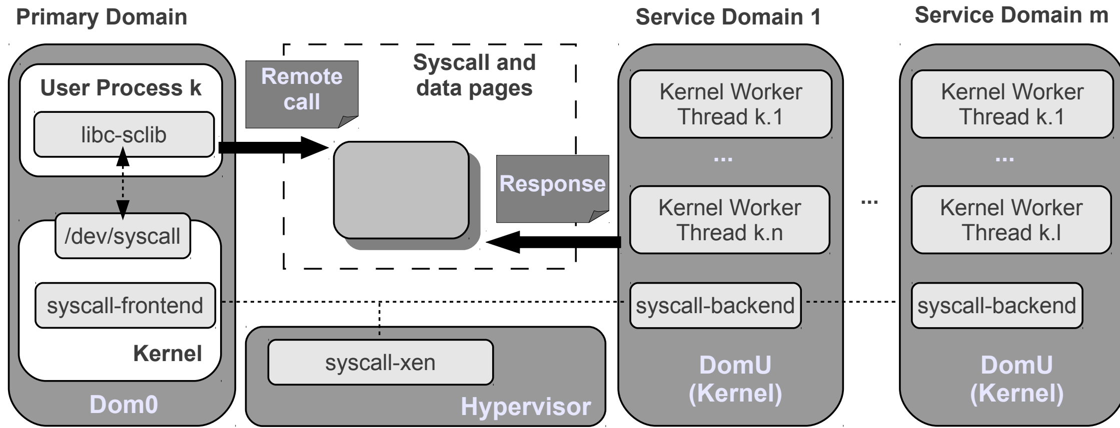

Figure 1: Architecture of VirtuOS

Our design does not assume that there is only one primary domain in which user processes run; it could be extended to support multiple user environments, each having its own primary and set of service domains. This makes VirtuOS's design applicable in traditional virtualization applications.

3.1 Failure Model

Our design goal is to contain faults originating in service domains only; we assume that the primary domain is stable enough to perform core system tasks such as task scheduling, IPC, and memory management. Service domains execute code that is potentially less reliable, such as drivers and corresponding software stacks.

VirtuOS provides recovery guarantees with respect to failures caused by software errors and transient hardware faults. Such hardware faults include invalid DMA memory accesses or interrupt signaling errors. Service domain failures can be contained as long as the hypervisor itself enforces isolation. We designed all communication between the primary domain and all service domains such that it can tolerate byzantine service domain failures, which implies careful handling of any requests or responses from those domains. If a failure is detected, the service domain must be restarted using standard hypervisor utilities. Service domain failures affect only those processes that have started using the failed domain; only these processes will need to be restarted.

We believe this model provides advantages compared to the alternative of rebooting the primary domain or the entire machine, especially when multiple service domains are used for different hardware components, such as separate network interfaces or storage devices, which may be accessed by disjoint subsets of processes. In addition, server applications such as web servers are often designed to use multiple OS processes, which can be restarted if failures occur.

3.2 System Call Design

VirtuOS processes communicate with service domains at the level of system calls. We refer to system calls destined for a service domain as remote system calls, whereas local system calls are directly handled by the primary domain. A modified C library contains all necessary infrastructure to transparently demultiplex local and remote system calls and forward remote calls to service domains. Since most programs and libraries do not execute system calls directly, this design enables source and binary compatibility with dynamically linked binaries.

Since most POSIX system calls use file descriptors, we tag file descriptors with their corresponding domain. As an example, a *socket(2)* system call for the `AF_INET*` families may be forwarded to the networking service domain, which creates a socket and assigns a file descriptor number in return. Any subsequent operation such as *read(2)* or *write(2)* will then be dispatched to the service domain from which it originated. To avoid the need for coordination between service domains and the primary domain in assigning file descriptor numbers, VirtuOS's C library translates user-visible file descriptors to domain file descriptors via a translation table. This design also allows the implementation of POSIX calls (e.g., *dup2()*) that assume that a process has control over its file descriptor space, but it requires that the user-level library interpose on all file descriptor related calls.

3.2.1 Exceptionless Dispatch

To dispatch system calls to a service domain, we initially considered the use of a traditional exception-based mechanism. We discarded this design option because every system call would then have required exiting the virtual machine in order to use an interdomain communication facility such as event channels, in addition to the costs associated with the mode switch itself. Instead, we adopted the concept of exceptionless system calls described in Section 2.2.

The implementation of exceptionless system calls across virtual machines poses a number of unique challenges that are not present when applying this method to optimize the performance of native system calls as done in FlexSC [48]. In FlexSC, the kernel worker threads handling the system calls can easily obtain direct access to a client thread's address space, file descriptor tables, credentials, and POSIX signal settings. Such direct access is impossible in VirtuOS since the kernel worker threads reside in a different virtual machine. Our implementation addresses these differences, which requires the primary domain to communicate essential information about running processes to the service domains, which we describe in Section 4.2.

A *front-end* driver in the primary domain kernel communicates with *back-end* drivers in the service domains to inform them when processes are created or have terminated. The front-end and back-end drivers also cooperate to establish the necessary shared memory areas between user processes and service domains. Each process creates two such shared areas for each service domain: (1) one area to hold the request queue for outstanding system call requests, and (2) an area used as a temporary buffer for system calls that transfer user data. We also added a component to the underlying hypervisor to keep track of service domain states and domain connection information, which is necessary for domain initialization and recovery after failures.

The request queue for outstanding system call requests consists of fixed-sized system call entries, which contain the information needed to execute a system call. System call entries are designed to be small so they fit into a single cache line. When a system call is dispatched to a service domain, a system call entry is added to the request queue shared with that domain. We adapted a FIFO/LIFO lock-free queue with ABA tagging [30,40,52] to ensure that the request queue, as well as other shared queues, can be accessed safely by both the user process and the service domain.

3.2.2 Managing User Memory Access

System call arguments may refer to user virtual addresses, such as when pointing to buffers a system call should copy into or out of. Our design uses a copying-based strategy in which the user process copies data into or out of a temporary buffer of memory shared with the service domain. A shared memory region is mapped in a continuous virtual address space region in the user program and in the service domain. During initialization, user programs use a pseudo */dev/syscall* device to create a memory mapping for this region. The primary domain's front-end driver, which services this device, then communicates to the back-end drivers within the service domains a request to allocate and grant access to pages that can be mapped into the user process's address space.

A special purpose allocator manages the allocation of buffers for individual system calls from this shared region. For simplicity, our implementation uses a simple explicit list allocator, along with a per-thread cache to reduce contention. If the free list does not contain a large enough buffer, the region can be dynamically grown via the */dev/syscall* device. The region can also be shrunk, although the allocator used in our current prototype does not make use of this facility. Since the POSIX API does not impose limits on the sizes of memory buffers referred to in system call arguments, we split large requests into multiple, smaller requests to avoid excessive growth.

Although this design for managing memory access requires an additional copy, it sidesteps the potential difficulties with designs that would provide a service domain with direct access to a user process's memory. Since a user process may provide any address in its virtual address space as an argument to a system call, direct access would require coordination with the primary domain's physical page management. Either pages would have to be pinned to ensure they remain in physical memory while a system call is in progress, which would severely restrict the primary domains flexibility in managing physical memory, or the primary domain would have to handle page faults triggered by accesses from the service domain, which would require complex and expensive interdomain communication.

3.2.3 Polling System Calls

Polling system calls such as *select(2)*, *poll(2)*, or Linux's *epoll_wait(2)* operate on sets of file descriptors that may belong to different domains. These calls block the current thread until any of these file descriptors change state, e.g. becomes readable, or until a timeout occurs. We implement these calls using a simple signaling protocol, in which the primary domain controls the necessary synchronization. We first partition the file descriptor set according to the descriptors' target domains. If all file descriptors reside within the same domain (local or remote), a single request to that domain is issued and no

interdomain coordination is required. Otherwise, we issue requests to each participating service domain to start the corresponding polling calls for its subset. Lastly, a local system call is issued that will block the current thread. If the local polling call completes first, the primary domain will issue notifications to all participating service domains to cancel the current call, which must be acknowledged by those domains before the call can return. If any remote call completes first, the corresponding service domain notifies the primary domain, which then interrupts the local call and starts notifying the other domains in the same manner as if it had completed first. The C library combines results from all domains before returning from the call.

3.3 Thread Management

VirtuOS uses separate strategies to schedule user-level threads issuing remote system call requests and to schedule worker kernel threads executing in service domains.

3.3.1 User-level Thread Scheduling

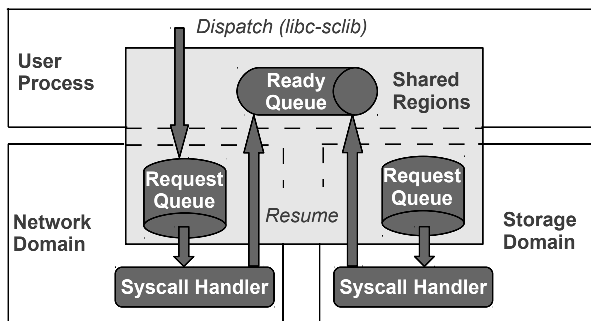

Figure 2: Sharing ready and request queues

To retain the performance benefits of exceptionless system call dispatch, we must minimize the synchronization costs involved in obtaining system call results. VirtuOS uses a combination of M:N user-level threading and adaptive spinning to avoid the use of exceptions when possible. The threading implementation uses a single, per-process ready queue, which resides in memory that is shared with all service domains. Like the per-service domain request queues, it is implemented in a lock-free fashion to allow race-free access from the service domains. Figure 2 shows the relationship between a process's ready queue and its per-domain requests queues. When a system call request is placed into a service domain's request queue, the issuing user-level thread includes a pointer to its thread control block in the system call entry. If other user-level threads are ready to execute, the current user-level thread blocks and performs a low-overhead context switch to the next ready user-level thread. Once the service domain completes the system call request, it directly accesses the user process's ready queue and resumes the blocked thread based on the pointer contained in the system call entry.

If there are no ready user-level threads after a system call request is issued, a user-level thread spins for a fixed amount of time, checking for either its system call request to complete or a new thread to arrive in the ready queue. Otherwise, it blocks the underlying kernel thread via a local system call, requiring an exception-based notification from the remote service domain when the request completes. Such spinning trades CPU capacity for latency. We determined the length of the spinning threshold empirically so as to maximize performance in our benchmarked workloads, as we will further discuss in Section 5.1. Alternative approaches include estimating the cost of exception-based notification in order to optimize the competitive ratio of the fixed-spin approach compared to an optimal off-line algorithm, or using an on-line algorithm based on sampling waiting times, as described in the literature [33].

Service domains perform range checking on any values read from the shared area in which the ready queue is kept, which is facilitated by the use of integer indices. Thus, although a failing service domain will affect processes that use it, sharing the ready queue will not cause faults to propagate across service domains. Though request queues are not shared across service domains, service domains must perform range checking when dequeuing system call requests to protect themselves from misbehaving processes; moreover, the domain subsequently subjects any arguments contained in these requests to the same sanity checks as in a regular kernel.

3.3.2 Worker Thread Scheduling

Each service domain creates worker threads to service system call requests. Our approach for managing worker threads attempts to maximize concurrency while minimizing latency, bounding CPU cost, maintaining fairness, and avoiding starvation.

We create worker threads on demand as system call requests are issued, but always maintain one spare worker thread per process. Once created, a worker thread remains dedicated to a particular process. This fixed assignment allows us to set up the thread's process-specific data structures only once. Although handling a system call request from a process must be serviced by a worker thread dedicated to that process, all worker threads cooperate in checking for new requests using the following strategy.

When a worker thread has completed servicing a system call, it checks the request queues of all other processes for incoming requests and wakes up worker

threads for any processes whose request queue has pending requests. Finally, it checks its own process's request queue and handles any pending requests. If no request is pending in any queue, the worker thread will continue to check those queues for a fixed spinning threshold. If the threshold is exceeded, the worker thread will block.

To avoid excessive CPU consumption due to having too many threads spinning, we also limit the number of worker threads that are spinning to be no larger than the number of virtual CPUs dedicated to the service domain. In addition, our design allows a service domain to eventually go idle when there are no requests for it. Before doing so, it will set a flag in the domain's state information page, which is accessible via a read-only mapping to user threads. User-level threads check this flag after adding system call requests to a domain's request queue, and initiate a local system call to wake up the service domain if needed. To avoid a race, the service domain will check its request queue one more time before going idle after it has set the flag.

4 Implementation

This section describes the specific implementation strategy used in our VirtuOS prototype, as well as difficulties and limitations we encountered.

4.1 Effort

We implemented our system based on the Linux 3.2.30 kernel and the Xen 4.2 hypervisor. We use the same kernel binary for the primary domain as for service domains. On the user side, we chose the uClibc library, which provides an alternative to the GNU C library (glibc). We selected uClibc after concluding that glibc's code generation approach and extensive system call inlining would make comprehensive system call interposition too difficult. Unlike when using exceptionless system calls for optimization, we require that all system calls, no matter at which call-site, are dispatched to the correct service domains. We replaced Linux's pthread library with our own to provide the M:N implementation described in Section 3.3.1.

Table 1 summarizes our implementation effort with respect to new or modified code. The relatively small number of changes needed to the Linux kernel shows that our virtualization-based approach enabled vertical slicing with comparably little effort.

Component	Number of Lines
Back-end/Front-end Driver	3115/2157
uClibc+NPTL/libaio	11152/2290
Linux kernel/Xen	1610/468
Total:	20792

Table 1: New or modified code

4.2 Service Domain Implementation

Service domains handle remote system calls by redirecting requests to the corresponding functions of the service domain kernel. To keep our implementation effort small, we reused existing facilities and data structures whenever possible. For each process, we create a shadow process control block (PCB) that keeps track of that process's environment, including its credentials and file descriptors, as well as additional information, such as the location of the regions containing system call arguments and data. The data structures contained in the shadow PCB track the data structures referenced by the process's PCB in the primary domain. For instance, the primary domain must keep the service domains' view of process's credentials and capabilities in sync. Any changes are propagated to all service domains, followed by a barrier before the next system call is executed.

We reuse the existing infrastructure to manage per-process file descriptor tables, although the shadow PCB's table includes only those file descriptors that were created by the service domain for a given process. Since VirtuOS's C library translates user visible file descriptors before issuing system call requests, a service domain can directly use the file descriptor numbers contained in system call arguments. Reusing the existing data structures also ensures the correct semantics for the *fork(2)* system call, which requires duplication of a process's file descriptor tables and credentials.

We exploit the existing mechanisms for the validation of user-provided addresses, i.e. the copy_from_user(), copy_to_user(), etc. functions that are already used in all system call handler functions when accessing memory. We set the executing worker thread's user memory range (via set_fs()) to accept only addresses within the shared region used for this purpose, as discussed in Section 3.2.2. Since all data pages are in contiguous regions in both the user process and the service domain, our system call library can pass addresses that are valid in the service domain as system call arguments, which are computed by applying an offset relative to the region's known base address.

4.3 Process Coordination

The primary and the service domains need to communicate when new processes that make use of a service domain are created or destroyed. The front-end driver in the primary domain informs the back-end driver of each

new process, which then performs initial memory setup and prepares for receiving system call requests.

The drivers communicate using Xen's interdomain communication facilities. We use Xen event channels as an interdomain interrupt facility and Xen I/O ring buffers to pass data between domains. We establish two separate event channels. The first one is used for adding and removing processes, as well as expanding and shrinking the shared memory regions used by processes to communicate with the service domain. We use a request/response protocol on this channel. We use a second event channel for all notifications coming to and from service domains. We use two separate ring buffers, one for each direction. We modified the ring buffer implementation to allow for one-directional communication (i.e., requests with no responses) to support asynchronous requests that do not require a synchronous response. Xen's ring buffers support batching so that only one interdomain interrupt is required when multiple requests or responses occur concurrently.

To ensure consistency, our back-end driver executes process management related requests from the front-end sequentially. For instance, a request to remove a process must be processed after the request to add that process. To avoid having to maintain dependencies between requests, we use a single-threaded implementation, which may create a bottleneck for some workloads.

We use two optimizations to speed up process management. First, we preallocate shared memory regions. When a new process is created, the front-end driver will attempt to use a preallocated region to map into the new process. If successful, it will issue a request to obtain grant references for the next region in anticipation of future processes. Otherwise, it must wait for the previous request to complete before installing the shared region. Second, we avoid waiting for responses where it is not necessary in order to continue, such as when shrinking the shared memory region.

4.4 Special Case Handling

Some system calls required special handling in VirtuOS. For example, the *ioctl(2)* and *fcntl(2)* system calls may contain a parameter that points to a memory region of variable size, depending on the request/command code passed. We handle fcntl() by treating each command code separately, performing any necessary copies. For ioctl(), we use the `_IOC_SIZE` and `_IOC_DIR` macros to decode memory argument size and direction, and include special handling for those ioctl() calls that do not follow this convention.

POSIX signals interact with system calls because receiving a signal may result in a system call interruption. To support this behavior, we inform service domains of pending signals for in-progress system calls. We mark the signal as pending in the remote worker thread's shadow process control block (PCB), resume the worker thread and let it abort the system call as if a local signal had been produced. In addition, we make note of pending signals through an additional flag in the system call request queue entries; this flag lets the worker thread recognize that a system call should be aborted even if the notification arrived before the request was started. To avoid spurious interruptions, we keep a per-thread counter that is incremented with each system call and store its current value in each system call request. The counter value functions as a nonce so that a service domain can match signal notifications to pending system calls and ignore delayed notifications.

The Linux /proc file system can be used to obtain information about file descriptors and other information pertaining to a process. Some applications inspect the information published via /proc. Our system call library translates accesses to /proc when a user process accesses the file descriptor directory so that programs see translated file descriptors only, hiding the fact that some file descriptors belong to service domains.

4.5 Limitations

The vertical decomposition of a monolithic kernel makes the implementation of calls that intertwine multiple subsystems difficult. For instance, Linux's *sendfile(2)* call, which directly transfers file data onto a network connection, must be implemented via user-level copying if the file and the network connection are serviced by different service domains.

Our prototype also does not support *mmap(2)* for file descriptors serviced by the storage domain. mmap could be supported by granting direct access to the service domain's memory, similar to how the request queue and data pages are shared. However, such an approach would limit the storage domain's flexibility in managing its page cache.

Our current prototype also does not provide transparency for file path resolution when a storage domain is involved. To recognize accesses to files stored in a storage domain, the C library keeps track of a process's current working directory and translates all relative paths to absolute ones. This approach provides a different semantics if a process's current working directory is removed.

5 Experimental Evaluation

Our experimental evaluation comprises of (1) an evaluation of VirtuOS's overhead during system call handling

Processor	2 x Intel Xeon E5520, 2.27GHz
Number of cores	4 per processor
HyperThreading	OFF (2 per core)
TurboBoost	OFF
L1/L2 cache	64K/256K per core
L3 cache	2 x 8MB
Main Memory	12 GB
Network	Gigabit Ethernet, PCI Express
Storage	SATA, HDD 7200RPM

Table 2: System configuration

and process coordination, (2) an evaluation of its performance for server workloads, and (3) a verification of its ability to recover from service domain failures.

Our goals are to show that VirtuOS imposes tolerable overhead for general workloads, that it retains the performance advantages of exceptionless system call dispatch for server workloads, and that it can successfully recover from service domain failures.

We run series of micro- and macrobenchmarks to that end. We use a native Linux system or Linux running inside a Xen domain as baselines of comparison, as appropriate. We were unable to compare to FlexSC due to its lack of availability.

Our current prototype implementation uses the Linux 3.2.30 kernel for all domains. We tested it with Alpine Linux 2.3.6, x86_64 (a Linux distribution which uses uClibc 0.9.33 as its standard C library) using a wide range of application binaries packaged with that distribution, including OpenSSH, Apache 2, mySQL, Firefox, links, lynx, and Busybox (which includes ping and other networking utilities). In addition, we tested compilation toolchains including GCC, make and abuild. Our system is sufficiently complete to be self-hosting.

We used the distribution-provided configurations for all programs we considered. Our system specification is shown in Table 2.

5.1 Overhead

Compared to a conventional operating system, VirtuOS imposes multiple sources of overhead, which includes file descriptor translation, spinning, signaling (if necessary), copying, and process coordination. All microbenchmarks were run at least 10 times; we report the average result. We found the results to be highly consistent, with a relative standard deviation of less or equal than 2%.

System Call Dispatch & Spinning. Our first microbenchmark repeatedly executes the *fcntl(2)* call to read a flag for a file descriptor that is maintained by the storage domain. In native Linux, this system call completes without blocking the calling thread. In VirtuOS, as described in Section 3.3.1, this call is submitted into the service domain's request queue. Our microbenchmark considers the single-threaded case in which there are no other user-level threads to run after the call is submitted. In this case, the calling thread will spin for a fixed number of iterations (the spinning threshold), checking if the request has been processed. If not, it will block the underlying kernel thread, requiring the more expensive, interdomain interrupt-based notification from the service domain.

For a single fcntl(2) call, we found that we needed to iterate at least 45 times for the system call to complete without blocking. If the system call completed without blocking, the achieved throughput was 0.7x that of the native case, which we attribute to the file translation and dispatch overhead, which outweighed any benefit due to exceptionless handling. Otherwise, if notification is necessary, we are experiencing a slowdown of roughly 14x. This result shows that spinning is a beneficial optimization for workloads that do not have sufficient concurrency to benefit from user-level threading. We found, however, that we needed a much larger spinning threshold (1000 iterations) to achieve the best performance for our macrobenchmarks. We use the same value for all benchmarks; on our machine, 1,000 iterations require approximately 26,500 machine cycles.

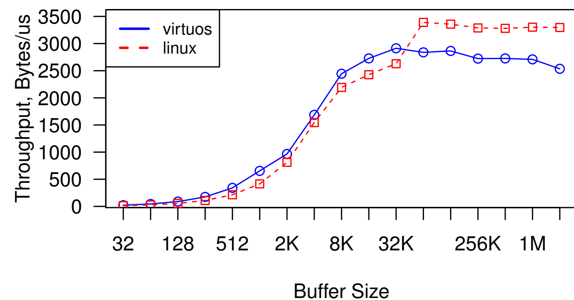

Figure 3: Absolute Throughput vs Buffer Size for writing to tmpfs

Copying Overhead. As described in Section 3.2.2, VirtuOS requires an additional copy for system calls that access user memory. Simultaneously, we expect those system calls to also benefit more greatly from exceptionless dispatch. We created a microbenchmark to evaluate these costs and benefits. The benchmark writes 16MB in chunks of 32, 64, up to 2MB to a file created in a *tmpfs* filesystem, which is provided by a native kernel in the baseline case and by a storage domain in VirtuOS. Linux implements this file system type using anonymous paged

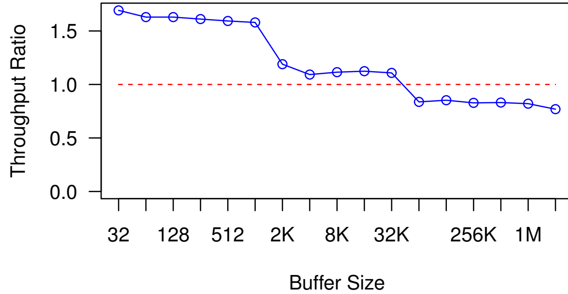

Figure 4: Throughput Ratio vs Buffer Size for writing to tmpfs

virtual memory. For the data sizes in our experiments, no paging is necessary. Thus, any differences reflect a combination of the effects of exceptionless system call handling and the additional data copy required by our method of dispatching system calls. Figure 3 shows absolute throughput for various block sizes, and Figure 4 shows the throughput ratio.

Since the overall amount of data written remains the same, smaller block sizes indicate a higher system call frequency. For block sizes less than 64K, the savings provided by the exceptionless model in VirtuOS outweigh the additional costs. Such small block sizes are common; as a point of reference, the file utilities included in our distribution's GNU coreutils package (e.g., cp & cat) use a 32K block size. For larger block sizes, the copying overhead becomes more dominant, reducing performance to about 0.8x that of native Linux.

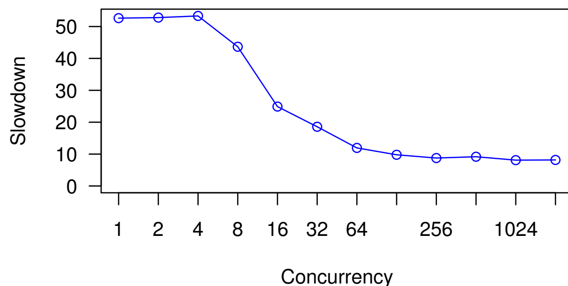

Figure 5: Process Creation Overhead

Process Coordination. As discussed in Section 4.3, the primary and the service domains need to coordinate when processes are created and destroyed. Requests to create and destroy processes are handled sequentially by a single kernel thread in each service domain. We created a microbenchmark that forks N concurrent processes, then waits for all of them. The forked processes simply exit. The case $N = 1$ represents the case of serial execution of single programs, such as in a shell script, albeit without actually executing any commands. Figure 5 shows a slowdown of 52x for this benchmark case, which decreases to about 8x as more processes are concurrently forked. This decrease shows the benefit of batching, which reduces the number of interdomain interrupts.

5.2 Performance

5.2.1 TCP Throughput and Latency

We first measured streaming TCP network performance by sending and receiving requests using the TTCP tool [1], using buffer sizes from 512 bytes to 16 KB.

We compared the performance of the following configurations (1) Unvirtualized native Linux; (2) Linux running in Xen/Dom0; (3) Linux running in Xen/DomU PVHVM with configured PCI passthrough for a network card; (4) Linux running in Xen/DomU using netback drivers in Dom0; (5) VirtuOS. We used configuration (4) as a proxy for the performance we would expect from a Xen driver domain, which we were unable to successfully configure with the most recent version of Xen.

For all configurations, we did not find any noticeable differences; all are able to fully utilize the 1 Gbps link with an achieved throughput of about 112.3 MB/s, independent of the buffer size used. In all configurations, this throughput is achieved with very low CPU utilization (between 1% and 6% for large and small buffer sizes, respectively) in Linux and VirtuOS's primary domain. We observed a CPU utilization of about 20% on the network service domain, due to the polling performed by kernel worker threads described in Section 3.3.2. This relative overhead is expected to decrease as the number of concurrent system call requests from the primary domain increases since more CPU time will be used for handling system call requests than for polling.

We also analyzed TCP latency using lmbench's *lat_tcp* tool [4], which measures the round-trip time for sending 1-byte requests. These results are shown in Figure 6. We used two spinning thresholds for VirtuOS: default and long, which correspond to the default settings used in our macrobenchmarks and to an indefinite threshold (i.e., spinning until the request completes).

Here, we observed that VirtuOS's latency is slightly higher than Linux's, but significantly less than when Xen's netfront/netback configuration is used. We conclude that VirtuOS performs better than alternative forms of driver isolation using Xen domains. Furthermore, if desired, its latency can be further reduced by choosing longer spinning thresholds, allowing users to trade CPU time for better latency.

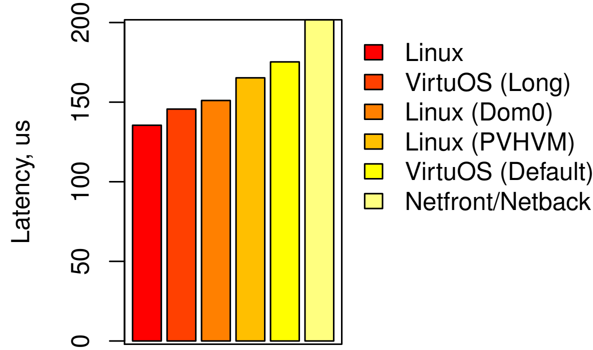

Figure 6: TCP 1-byte roundtrip latency (lat_tcp)

5.2.2 Multithreaded programs

We evaluated the performance of multithreaded programs when using the network and storage domains. We use the OLTP/SysBench macrobenchmark [5] to evaluate the performance of VirtuOS's network domain. In this benchmark, a mySQL server running in VirtuOS receives and responds to 10,000 requests, each comprising of 10 selection queries with output ordering, sent by network clients. The client uses multiple, concurrent threads, each of which issues requests sequentially. The files containing the database are preloaded into the buffer cache to avoid skewing the results by disk I/O. The benchmark records the average response time over all requests; we report throughput computed as *number of threads / average response time*.

We present throughput and the throughput ratio of VirtuOS vs Linux in Figures 7 and 8, respectively. VirtuOS's performance in this benchmark mostly matches or exceeds that of Linux by 1-16%.

To evaluate the performance of the storage domain, we used the FileIO/SysBench benchmark [5]. This benchmark generates 128 files with 1GB of total data and performs random reads with a block size of 16KB. We examined two configurations. In the first configuration, shown in Figure 9, we eliminated the influence of actual disk accesses by ensuring that all file data and metadata was kept into the buffer cache. In the resulting memory bound configuration, we observed between 30% and 40% performance loss, which we attribute to the cost of the additional memory copy. Compared to the microbenchmark presented in Section 5.1, the use of many concurrent threads exerts higher pressure on the L1/L2 cache, which increases the copying overhead. As part of our future work, we plan to verify this hypothesis using hardware performance counters, which will require the adaptation of a performance counter framework developed in prior work [42]. Figure 10 shows the relative performance for a mixed workload that includes random reads and random writes. Here we allow the Linux kernel and the storage domain to pursue their usual write back policies. Both systems provide roughly similar performance in this case.

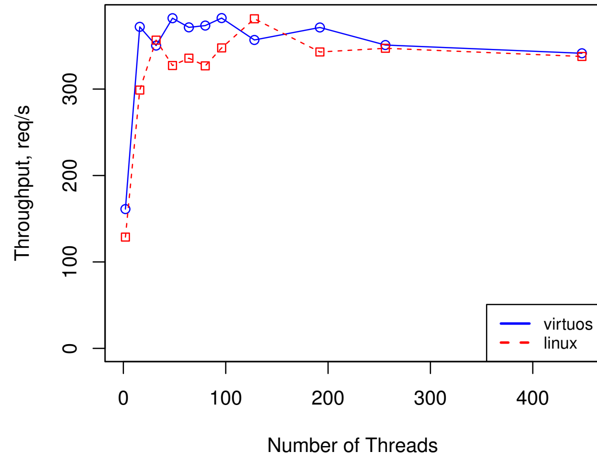

Figure 7: OLTP/SysBench mySQL throughput

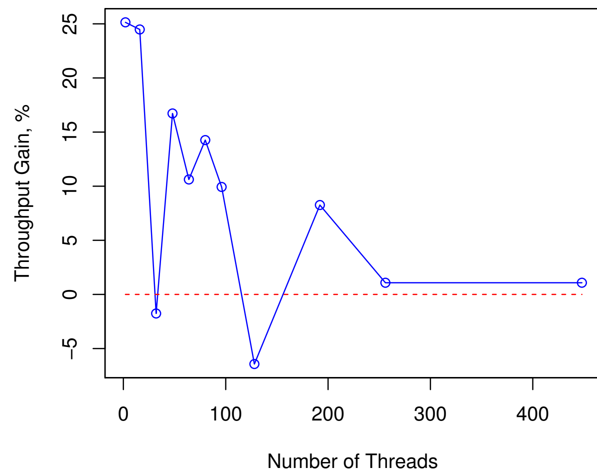

Figure 8: OLTP/SysBench mySQL throughput gain

Taken together, these benchmarks shows that for multithreaded workloads which benefit from M:N threading, it is possible to achieve performance that is at least as good as native Linux's.

5.2.3 Multiprocess programs

We also tested VirtuOS with single-threaded, multiple process applications such as Apache 2, and compared performance with native Linux. Single-threaded applications cannot directly utilize benefits of the M:N thread model and, hence, may require notification if system calls do not complete within the spinning threshold.

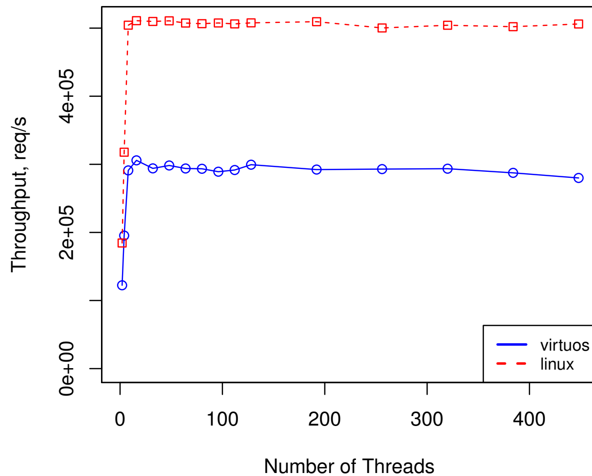

Figure 9: FileIO/SysBench throughput without disk accesses

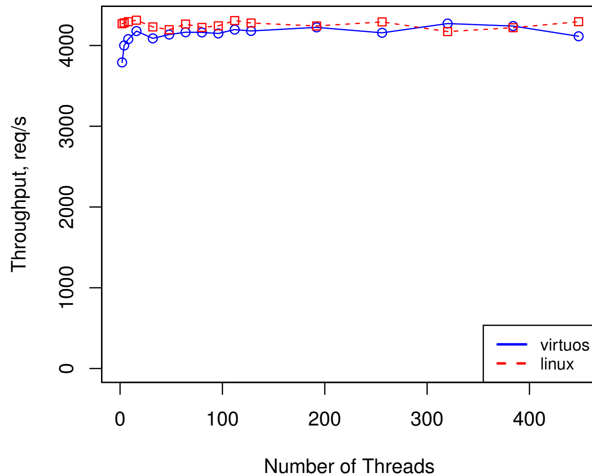

Figure 10: FileIO/SysBench throughput with disk accesses

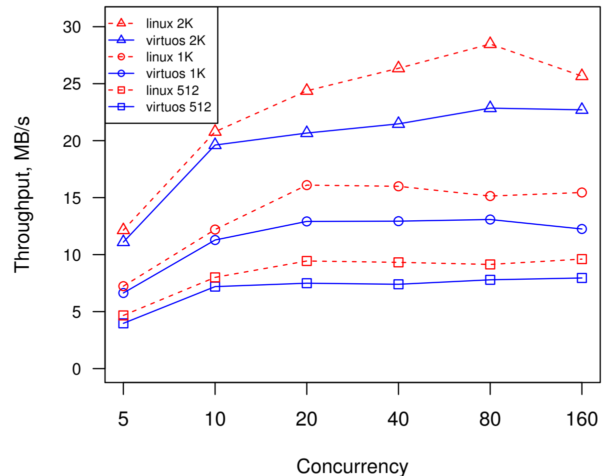

Figure 11: Apache 2 throughput for small objects

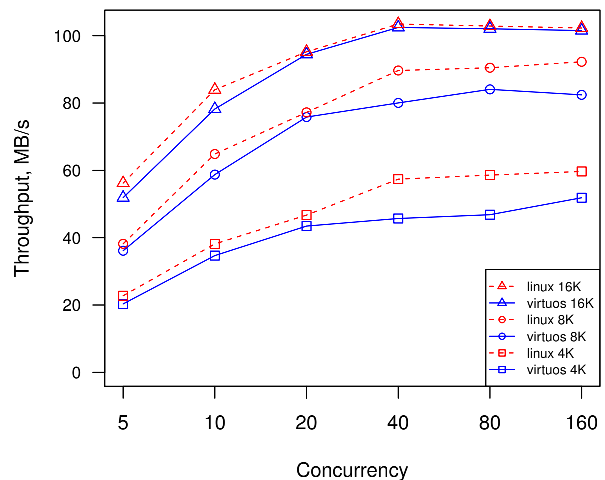

Figure 12: Apache 2 throughput for large objects

We used the Apache benchmark utility (*ab*) to record throughput while retrieving objects of various sizes. We present throughput for small and large objects in Figures 11 and 12. Both Linux and VirtuOS are able to saturate the outgoing Gigabit link for objects >= 16KB in size; for smaller sizes, VirtuOS's performance lags that of Linux by up to 20%. Adjusting the spinning threshold in either direction did not improve those numbers.

5.2.4 Concurrency Tunables

We found that the results in sections 5.2.2 to 5.2.3 are sensitive to choosing multiple parameters correctly. Table 3 shows the assignment that worked best for our benchmarks. We found that we needed to provide as many VCPUs to the primary domain as there are physical cores (8) for many tests, except for the FileIO benchmarks, where we needed to limit to number of VCPUs available to the primary domain to ensure enough CPU capacity for the service domain. The optimal number of VCPUs assigned to the respective service domain varied by benchmark. We let the Xen scheduler decide on which cores to place those VCPUs because we found that pinning VCPUs to cores did not result in higher performance, except for the FileIO benchmarks, where assigning the VCPUs of the primary domain to the cores of one physical CPU and the VCPUs of the service domain to the other resulted in approx. 15% higher throughput. For our Linux baseline, we used all 8 cores.

In addition, we observed that it is also beneficial to tune the maximum number of kernel threads created by our M:N library for the multi-threaded workloads. We note that such a limit may lead to deadlock if all available kernel threads are blocked in local system calls and

Program	Pri Domain	Service Domain	M:N
mySQL	8 VCPUs	1 VCPU	M:18
FileIO	4 VCPUs	4 VCPUs	M:4
Apache	8 VCPUs	3-4 VCPUs	N/A

Table 3: Number of service domain VCPUs and kernel threads

the threading library does not create new kernel threads on demand, which our current prototype does not implement.

5.3 Failure Recovery

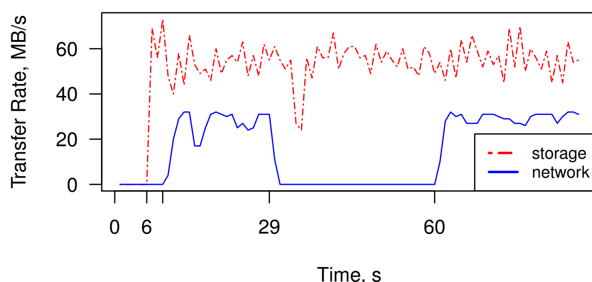

Figure 13: Failure recovery scenario

VirtuOS supports failure recovery for any faults occurring in service domains, including memory access violations, interrupt handling routine failure and deadlocks. Such faults may cause the affected domain to reboot; otherwise, the domain must be terminated and restarted using Xen's toolkit utilities. We provide a cleanup utility to unregister terminated service domains, free any resources the primary domain has associated with them, and unblock any threads waiting for a response from the service domain. We do not currently provide a fault detector to detect when a domain should be restarted. Our recovery time is largely dominated by the amount of time it takes to reboot the service domain (approx. 20 seconds in our current configuration).

We designed an experiment to demonstrate that (1) a failure of one service domain does not affect programs that use another one; (2) the primary domain remains viable, and it is possible to restart affected programs and domains. In this scenario, we run the Apache server which uses the network domain. A remote client connects to the server and continually retrieves objects from it while recording the number of bytes transferred per second. To utilize the storage domain, we launch the Unix `dd` command to sequentially write to a file. We record the number of bytes written per second by observing the increase used disk space during the same second interval.

Figure 13 shows the corresponding transfer rates. At instant 0, the Apache server is launched. At instant 6, the `dd` command starts writing data to the disk. At instant 9, a remote client connects to the Apache server and starts using it. At instant 29, the network domain is abruptly terminated, reducing the client's observed transfer rate to 0, without affecting the ability of `dd` to use the storage domain. At instant 60, the network domain and the Apache server are restarted, and the remote client continues transferring data.

6 Related Work

VirtuOS's goal of decomposing kernel functionality is shared with several other approaches. Microkernels such as Mach [6] and L4 [37] provide an architecture in which only essential functionality such as task scheduling and message-based interprocess communication is implemented inside the kernel, whereas most other system components, including device drivers, are implemented in separate user processes. Aside from optimizing IPC performance, microkernel-based systems often devote substantial effort to creating compatibility layers for the existing system APIs, e.g. POSIX [27]. Multiserver operating system designs such as Sawmill [12, 22, 50] pursue the opposite approach by attempting to deconstruct a monolithic kernel's functionality into separate servers running on top of a microkernel. Multiserver OSs differ from VirtuOS in the methods used for communication and protection. Moreover, VirtuOS does not currently attempt to address all goals multiserver OSs address, such as supporting system-wide policies or resource sharing [22].

Multiple approaches exist to improve the robustness and reliability of device drivers and systems code. Nooks [51] introduced hardware protection domains inside a monolithic kernel to isolate device drivers from each other and from the remaining kernel. Such isolation protects against buggy drivers that may perform illegal memory accesses. Nooks demonstrated how to restructure an existing kernel's interaction with its drivers to facilitate the use of intrakernel protection domains, and explored the trade-off between benefits due to isolation and costs imposed by the domain crossings this approach requires.

Microdrivers [21] divide driver code to isolate hardware-specific and performance critical parts, which are run inside the kernel, from the remaining majority of the code which is run as a user process and can be written in a higher-level language [44]. Mainstream OSs have provided support for writing device drivers that execute in user mode for some time, but these fa-

cilities have not been widely used because the added context switches made it difficult to achieve good performance [34]. Some systems provide the ability to run unchanged drivers in user mode. DD/OS [35] provides this ability by creating a virtual machine built as a user-level task running on top of L4, whereas SUD [10] provides such an environment inside ordinary Linux user processes.

SafeDrive [57] uses a type system approach to provide fine-grained isolation of kernel components written in C, which relies upon a combination of compile-time analysis and runtime checking. Carburizer [31] analyzes and modifies driver code to withstand hardware failures by removing the assumption that the underlying hardware behaves according to its specification. Static analysis approaches have been used to find bugs and improve the reliability of systems code [7, 17], as well as approaches derived from model checking [55]. Domain-specific languages can reduce race conditions, deadlocks and protocol violations by formally describing the underlying hardware's behavior (e.g., Devil [39]) or the driver's expected software interface (e.g., Dingo [46]).

Multiple systems deploy exceptionless techniques: FlexSC [48, 49] proposed the use of exceptionless system calls, which we adopted in VirtuOS, for the purposes of optimizing system call performance in a monolithic kernel. Exceptionless communication techniques are also used in the fos [53] and Corey systems [9]. Both of these systems are designed for scalability on multicore systems and distribute OS services across cores. VirtuOS shares with these systems the assumption that the increasing availability of cores makes their dedication for systems-related tasks beneficial.

VirtuOS relies on an underlying hypervisor, and could benefit from a number of orthogonal ideas that were introduced to improve virtual machine technology. For instance, self-virtualizing hardware [43] makes it easier to safely and efficiently multiplex devices across domains, which would allow multiple service domains to share a device. Spinlock-aware scheduling [56] modifies the hypervisor's scheduler to avoid descheduling VCPUs that execute code inside a critical region protected by a spinlock, which could adversely affect the performance of service domains. The Fido system [11] optimizes Xen's interdomain communication facilities by allowing read-only data mappings to enable zero-copy communication if the application's trust assumptions allow this. The Xoar system [14] addresses the problems of improving manageability and robustness by splitting a control VM (i.e., Dom0 in Xen) into multiple, individually restartable VMs.

A number of systems attempt to minimize the impact of failures of isolated components and to speed up recovery after failures. The microreboot approach [13] advocates designing server applications as loosely coupled, well-isolated, stateless components, which keep important application state in specialized state stores. In doing so, individual components can be quickly restarted with limited loss of data. CuriOS [15] applies similar ideas to a microkernel-based OS. VirtuOS's use of existing kernel code in its service domains prevents us from using this approach since monolithic kernels make free and extensive use of shared data structures. Fine-grained fault tolerance (FGFT [32]) uses device state checkpoints to restore the functionality of drivers after a failure, but it so far has been applied only to individual drivers rather than the state of an entire kernel.

7 Conclusion

This paper presented VirtuOS, a fault-resilient operating system design which provides isolation for kernel components by running them in virtualized service domains. Service domains are constructed by carving a vertical slice out of an existing Linux kernel for a particular service, such as networking or storage.

VirtuOS allows processes to directly communicate with service domains via exceptionless system call dispatch. Thus, user processes can transparently benefit from isolated service domains without requiring the use of a special API. A special-purpose user-level threading library allows service domains to efficiently resume threads upon system call completion via direct queue manipulation. To the best of our knowledge, VirtuOS is the first system to use virtual machines for system call dispatch and to apply exceptionless communication across virtual machines.

We have tested our VirtuOS prototype with several existing applications; our experimental evaluation has shown that our design has the potential to outperform not only existing solutions for driver isolation but can for concurrent workloads that benefit from M:N threading meet and exceed the performance of a traditional, monolithic Linux system using exception-based system call dispatch.

Availability. VirtuOS's source code is available at `http://people.cs.vt.edu/~rnikola/` under various open source licenses.

Acknowledgments. We would like to thank the anonymous reviewers and our shepherd Kevin Elphinstone for their insightful comments and suggestions, which helped greatly improve this paper.

References

[1] TTCP tool. http://www.netcore.fi/pekkas/linux/ipv6/ttcp.c, 2007.

[2] Kernel Asynchronous I/O (AIO). http://lse.sourceforge.net/io/aio.html, 2012.

[3] uClibc C libary. http://uclibc.org/, 2012.

[4] LMbench – Tools for Performance Analysis. http://lmbench.sourceforge.net/, 2013.

[5] SysBench 0.4.12 – A System Performance Benchmark. http://sysbench.sourceforge.net/, 2013.

[6] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–112, 1986.

[7] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems*, EuroSys'06, pages 73–85, Leuven, Belgium, 2006.

[8] P. Barham, B. Dragovic, K. Fraser, and et al. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 164–177, Bolton Landing, NY, USA, 2003.

[9] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'08, pages 43–57, San Diego, CA, 2008.

[10] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, ATC'10, pages 117–130, Boston, MA, USA, 2010.

[11] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson. Fido: fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 USENIX Annual Technical Conference*, ATC'09, pages 313–326, San Diego, CA, USA, 2009.

[12] T. Bushnell. Towards a new strategy for OS design, 1996. http://www.gnu.org/software/hurd/hurd-paper.html.

[13] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'04, pages 31–44, San Francisco, CA, USA, 2004.

[14] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, pages 189–202, Cascais, Portugal, 2011.

[15] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: improving reliability through operating system structure. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'08, pages 59–72, San Diego, CA, USA, 2008.

[16] U. Drepper and I. Molnar. The native POSIX thread library for Linux, 2005. http://www.akkadia.org/drepper/nptl-design.pdf.

[17] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP'01, pages 57–72, Banff, Alberta, Canada, 2001.

[18] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'96, pages 137–151, Seattle, WA, USA, 1996.

[19] K. Fraser, H. Steven, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on-demand IT InfraStructure*, OASIS'04, 2004.

[20] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th Conference on Large Installation System Administration*, LISA '06, pages 149–159, Washington, DC, USA, 2006.

[21] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 168–178, Seattle, WA, USA, 2008.

[22] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 109–114, Kolding Denmark, 2000.

[23] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, Big Sky, MT, USA, 2009.

[24] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, HOTOS'05, Santa Fe, NM, 2005.

[25] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP'97, pages 66–77, Saint Malo, France, 1997.

[26] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *SIGOPS Operating Systems Review*, 40(1):95–99, Jan. 2006.

[27] J. Helander. Unix under Mach: The Lites Server. Master's thesis, Helsinki University of Technology, 1994.

[28] J. Herder, D. Moolenbroek, R. Appuswamy, B. Wu, B. Gras, and A. Tanenbaum. Dealing with driver failures in the storage stack. In *Proceedings of the 4th Latin-American Symposium on Dependable Computing*, LADC'09, pages 119–126, Joao Pessoa, Brazil, 2009.

[29] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. The architecture of a fault-resilient operating system. In *Proceedings of 12th ASCI Conference*, ASCI'06, pages 74–81, Lommel, Belgium, 2006.

[30] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[31] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP'09, pages 59–72, Big Sky, MT, USA, 2009.

[32] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 473–484, Houston, Texas, USA, 2013.

[33] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, SOSP'91, pages 41–55, Pacific Grove, CA, USA, 1991.

[34] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. Userlevel device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sept. 2005.

[35] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'04, pages 17–30, San Francisco, CA, USA, 2004.

[36] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP'93, pages 175–188, Asheville, NC, USA, 1993.

[37] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP'95, pages 237–250, Copper Mountain, CO, USA, 1995.

[38] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'05, pages 13–23, Chicago, IL, USA, 2005.

[39] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: an IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'00, pages 17–30, San Diego, CA, USA, 2000.

[40] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, PODC'96, pages 267–275, Philadelphia, PA, USA, 1996.

[41] B. Murphy. Automating software failure reporting. *Queue*, 2(8):42–48, Nov. 2004.

[42] R. Nikolaev and G. Back. Perfctr-Xen: a framework for performance counter virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'11, pages 15–26, Newport Beach, CA, USA, 2011.

[43] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC'07, pages 179–188, Monterey, CA, USA, 2007.

[44] M. J. Renzelmann and M. M. Swift. Decaf: moving device drivers to a modern language. In *Proceedings of the 2009 USENIX Annual Technical Conference*, ATC'09, pages 187–200, San Diego, CA, USA, 2009.

[45] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 129–144, 2000.

[46] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: taming device drivers. In *Proceedings of the 4th ACM European Conference on Computer systems*, EuroSys'09, pages 275–288, Nuremberg, Germany, 2009.

[47] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC'08, pages 29–42, Boston, Massachusetts, 2008.

[48] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'10, pages 1–8, Vancouver, BC, Canada, 2010.

[49] L. Soares and M. Stumm. Exception-less system calls for event-driven servers. In *Proceedings of the 2011 USENIX Annual Technical Conference*, ATC'11, pages 131–144, Portland, OR, 2011.

[50] J. M. Stevenson and D. P. Julin. Mach-US: UNIX on generic OS object servers. In *Proceedings of the USENIX 1995 Technical Conference*, TCON'95, pages 119–130, New Orleans, Louisiana, 1995.

[51] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 207–222, Bolton Landing, NY, USA, 2003.

[52] R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

[53] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, Apr. 2009.

[54] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'02, pages 195–209, Boston, MA, USA, 2002.

[55] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'06, pages 131–146, Seattle, WA, USA, 2006.

[56] A. Zhong, H. Jin, S. Wu, X. Shi, and W. Gen. Optimizing Xen hypervisor by using lock-aware scheduling. In *Proceedings of the Second International Conference on Cloud and Green Computing*, CGC'2012, pages 31–38, 2012.

[57] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'06, pages 45–60, Seattle, WA, USA, 2006.