Operating System Virtualization: Practice and Experience

Oren Laadan
Computer Science Department
Columbia University
New York, NY, 10027
orenl@cs.columbia.edu

Jason Nieh
Computer Science Department
Columbia University
New York, NY, 10027
nieh@cs.columbia.edu

ABSTRACT

Operating system (OS) virtualization can provide a number of important benefits, including transparent migration of applications, server consolidation, online OS maintenance, and enhanced system security. However, the construction of such a system presents a myriad of challenges, even for the most cautious developer, that if overlooked may result in a weak, incomplete virtualization. We present a detailed discussion of key implementation issues in providing OS virtualization in a commodity OS, including system call interposition, virtualization state management, and race conditions. We discuss our experiences in implementing such functionality across two major versions of Linux entirely in a loadable kernel module without any kernel modification. We present experimental results on both uniprocessor and multiprocessor systems that demonstrate the ability of our approach to provide fine-grain virtualization with very low overhead.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms

Design, Experimentation, Performance

Keywords

Operating Systems, Virtualization

1. INTRODUCTION

Computers have become ubiquitous in academic, corporate, and government organizations as exponential scaling laws have made computers faster, cheaper, and increasingly connected. At the same time, the widespread use of computers has given rise to enormous management complexity and security hazards. Virtualization has emerged as a key technology for addressing these issues.

Virtualization essentially introduces a level of indirection to a system to decouple applications from the underlying host system. This decoupling can be leveraged to provide important properties such as isolation and mobility, providing a myriad of useful benefits. These benefits include supporting server consolidation by isolating applications from one another while sharing the same machine, improved system security by isolating vulnerable applications from other mission critical applications running on the same machine, fault resilience by migrating applications off of faulty hosts, dynamic load balancing by migrating applications to less loaded hosts, and improved service availability and administration by migrating applications before host maintenance so that they can continue to run with minimal downtime.

While virtualization can be performed at a number of different levels of abstraction, providing virtualization at the correct level to transparently support unmodified applications is crucial in practice to enable deployment and widespread use. The two main approaches for providing application transparent virtualization are hardware virtualization and operating system (OS) virtualization. Hardware virtualization techniques [3, 29, 33, 36] virtualize the underlying hardware architecture using a virtual machine monitor to decouple the OS from the hardware so that an entire OS environment and associated applications can be executed in a virtualized environment. OS virtualization techniques [6, 24, 27, 30, 32, 34, 37] virtualize the OS to decouple applications from the OS so that individual applications can be executed in virtualized environments. Hardware and OS virtualization techniques each provide their own benefits and can provide complementary functionality.

OS virtualization provides a fine granularity of control at the level of individual processes or applications, which is more beneficial than the hardware virtualization abstraction that works with entire OS instances. For example, OS virtualization can enable transparent migration of individual applications, not just migration of entire OS instances. This finer-granularity migration provides greater flexibility and results in lower overhead [21, 24]. Furthermore, if the operating system requires maintenance, OS virtualization can be used to migrate the critical applications to another running operating system instance. By decoupling applications from the OS instance, OS virtualization enables the underlying OS to be patched and updated in a timely manner with minimal impact on the availability of application services [26]. Hardware virtualization alone cannot provide this functionality since it ties applications to an OS instance, and commodity operating systems inevitably incur downtime due to necessary maintenance and security updates.

Given the benefits of OS virtualization, contemporary OSs are increasingly interested in providing support for it [5]. While many of the concepts behind OS virtualization have been discussed in detail in previous work, little attention has been given to understanding how to actually implement it in

SYSTOR 2010 May 24–26, Haifa, Israel

practice. Most work has focused on higher-level issues without regard for many of the subtle issues and implementation challenges in making OS virtualization function correctly for unmodified applications and commodity OSs. While some work has focused on higher level implementation considerations regarding security in OS virtualization [11], we are not aware of any previous work that considers implementation issues in providing more complete OS virtualization, such as in the context of transparent application migration.

We present a detailed discussion of key implementation issues and challenges in providing OS virtualization in a commodity OS. We compare alternatives for implementing OS virtualization at user-level vs. kernel-level, discuss performance costs for methods of storing virtualization state, and examine subtle race conditions that can arise in implementing OS virtualization. Some OSs are gradually incorporating virtualization support by making pervasive changes to the OS kernel [22]. We describe an approach of implementing OS virtualization in a minimally invasive manner by treating the OS kernel as an unmodified black box. The experiences from this approach are instrumental in demonstrating how OS virtualization can be incorporated into commodity OSs with minimal changes. Using this approach, we have implemented a Linux OS virtualization prototype entirely in a loadable kernel module. We present quantitative results demonstrating that such a minimally invasive approach can be done with very low overhead.

2. VIRTUALIZATION CONCEPTS

OS virtualization isolates processes within a virtual execution environment by monitoring their interaction with the underlying OS instance. Similar to hardware virtualization [25], applications that run within the virtual environment should exhibit an effect identical to that demonstrated as if they had been run on the unvirtualized system. In addition, a statistically dominant subset of the applications interaction with system resources should be direct to minimize overhead.

We classify OS virtualization approaches along two dimensions, host-independence and completeness. Host-dependent virtualization only isolates processes while host-independent virtualization also decouples them. The distinction is that host-dependent virtualization simply blocks or filters out the namespace between processes, while host-independent virtualization provides a private virtual namespace for the applications' referenced OS resources. The former does not support transparent application migration since the lack of resource translation tables mandates that the resource identifiers of an application remain static across hosts for a migrating process, which can lead to identifier conflicts when migrating between hosts. Examples of host-dependent virtualization include Linux VServers [34] and Solaris Zones [27]. Host-independent virtualization encapsulates processes in a private namespace that translates resource identifiers from any host to the private identifiers expected by the migrating application. Examples of this approach include Zap [21, 24] and Capsules [30]. We refer to this virtual private namespace as a *pod*, based on the terminology used in Zap.

In terms of completeness, partial virtualization virtualizes only a subset of OS resources. The most common example of this is virtual memory, which provides each process with its own private memory namespace but doesn't virtualize any other OS resources. As another example, the FreeBSD Jail [19] abstraction provides partial virtualization by restricting access to the filesystem, network, and processes outside of the jail, but does not regulate SysV interprocess communication (IPC) mechanisms. While partial virtualization has been used to support tighter models of security by limiting the scope of faulty or malicious processes, it can be unsafe if there exist direct or indirect paths for processes inside the environment to access resources outside or even break out of the environment. The `chroot` environment in Unix is a notorious example of a filesystem partial virtualization mechanism that has serious security shortcomings [7].

Subsystem	Description
Process ID	PID and related IDs: thread group, process group, session
Filesystem	Filesystem root (`chroot`)
SysV IPC	ID and KEY of message queues, semaphores, and shared memory
Unix IPC	Unix domain sockets, pipes, named pipes
Network	Internet domain sockets
Devices	Device specific resources
Pseudo terminals	PTS IDs and `devpts` pseudo filesystem
Pseudo filesystems	E.g. `procfs`, `devpts`, `shmfs`
Miscellaneous	Hostname, user/group ID, system name

Table 1: Kernel subsystems and related resources

Complete virtualization virtualizes all OS resources. While commodity OSs provide virtualization for some resources, complete virtualization requires virtualization for many resources that are already not virtualized, including process identifiers (PIDs), keys and identifiers for IPC mechanisms such as semaphores, shared memory, and message queues, and network addresses. Table 1 provides a summary of these additional resources that must be virtualized; a more detailed discussion is presented in [24].

Within this taxonomy of virtualization approaches, complete and host-independent virtualization provides the broadest range of functionality, which includes providing the necessary support for both isolation and migration of applications. An additional distinction between the taxonomies is in the scope of the application with respect to the available systems. Virtualization approaches that are host-dependent and/or partial provide benefits only on a single host, while complete, host-independent virtualization approaches provide the support for applications to exploit the available systems that are accessible to the entire organization. The remainder of this paper focuses on the demands of supporting this more general form of virtualization in the context of commodity OSs.

3. INTERPOSITION ARCHITECTURE

To support private virtual namespaces, mechanisms must be provided to translate between the pod's resource identifiers and the operating system resource identifiers. For every resource accessed by a process in a pod, the virtualization layer associates a *virtual name* to an appropriate OS *physical name*. When an OS resource is created for a process in a pod, the physical name returned by the system is caught, and a corresponding private virtual name is created and returned to the process. Similarly, any time a process passes a virtual name to the operating system, the virtualization layer catches and replaces it with the corresponding physical name. To enable this translation, a mechanism must be employed that redirects the normal control flow of the system so that the private virtual namespaces are employed rather than the default physical namespace.

Method	Description
System-wide hash table	Convert physical host identifiers to virtual pod identifiers
Per-pod hash table	Convert virtual pod identifiers to physical host identifiers
Direct reference	Per-process fast reference to augmented virtualization state
PID reference count	Protect PIDs of processes that insides or outside pods from reuse
`in-pod` process flag	Indicate that a process is running inside a pod
`init-pending` process flag	Indicate that a process in a pod is pending initialization
Outside-pod table	Track identifiers used by processes running outside pods
Restricted-ID table	Track identifiers without a reference count that are in use
`init-complete` process flag	Indicate that the virtualization state of a resource has been initialized
Filesystem stacking	Virtualize per-pod pseudo filesystems view

Table 2: Summary of virtualization methods

Interposition is the key mechanism that can provide the requisite redirection needed for virtualization of namespaces. In our context, interposition captures events at the interface between applications and the OS and performs some processing on those events before passing them down to the OS or up to the applications. The interposition that needs to be done for implementing OS virtualization requires that some preprocessing be done before the native kernel functionality is executed, and some post-processing be done after the native kernel functionality is executed. The interposition implementation itself is accomplished by wrapping the existing system calls with our functions and translating between virtual names and physical names before and after the original system call is invoked.

System call interposition can be implemented at different layers of the system. We advocate using the loadable kernel module technology that is now available with all major commodity OSs. A kernel module can provide application-transparent virtualization without base kernel changes and without sacrificing scalability and performance. In addition, by operating in privileged mode, virtualization can provide the security necessary to ensure correct isolation. By working at the level of kernel modules, the virtualization module can utilize the set of exported kernel subroutines, which is a well-defined interface. Using the kernel API also denotes a certain level of portability and stability in the implementation since changes in the kernel API are infrequent. In other words, virtualization portability is protected to a large extent from kernel changes in a similar way as legacy applications are protected.

There are other approaches to implementing system call interposition. One approach is to implement interposition as a user-level library [18, 20] such that interposition code is executed in the context of the process executing the system call. This is relatively easy to implement, potentially yields more portable code, and utilizes the clear boundary between user-level and kernel-level. Unfortunately, it does not provide effective isolation of applications and can be easily subverted at any time. It instead requires their cooperation and does not work for statically-linked libraries or directly executed system calls.

Another approach is to use a kernel process tracing facility such as `ptrace` [23], which allows a user-level process to monitor another process [35]. By using available kernel functionality, this process tracing approach can enforce an OS virtualization abstraction more effectively than strictly user-level approaches. However, `ptrace` has many limitations in terms of performance and security [35], and the semantics of `ptrace` are highly system-specific, which results in a non-portable method.

A third approach is to modify the kernel directly to implement interposition. This offers maximum flexibility, with the lowest interposition overhead. However, writing code directly in the kernel is more complicated and cumbersome than in user-level, harder to debug, and the result is most likely to be non-portable. Tying the implementation to the kernel internals requires tracking, in detail, all subsequent kernel updates. Furthermore, imposing a kernel patch, recompilation and reboot process is a serious practical barrier to deployment and ease-of-use.

Given the limitations of other approaches, we have implemented OS virtualization as a loadable kernel module that works with major Linux kernel versions, including both Linux 2.4 and 2.6 kernels. Our implementation avoids modifications to the operating system kernel, and aims to build strictly on its exported interface as much as possible. It supports the pod abstraction but also allows other processes to run outside of virtualized environments to ease deployment on systems which require such legacy functionality.

4. VIRTUALIZATION CHALLENGES

Given this kernel module interposition architecture, we now discuss key implementation challenges in supporting virtualized system calls. Virtualization requires that some state be maintained by the virtualization module. The basic state that needs to be maintained is the pod's resource names, the underlying system physical resource names, and the mapping between virtual and physical names. Throughout this discussion we emphasize that performance is a primary concern and many of our approaches are engineered to achieve low performance overhead. Table 2 provides a summary of the methods and data structures used to maintain virtualization state efficiently.

A first approximation approach employs two types of hash tables that can be quickly indexed to perform the necessary translation. One is a system-wide hash table indexed by physical identifiers on the host OS, that returns the corresponding pod and virtual identifier. The other is a per-pod hash table indexed by virtual identifiers specific to a pod, that returns the corresponding physical identifiers. A separate pair of hash tables would be used for each OS resource that needs to be virtualized, including PIDs, SysV IPC, and pseudo terminals. For multiprocessor and multi-threaded systems, proper hash table maintenance requires locking mechanisms to ensure state consistency. Handling these locks to avoid deadlock and to lower performance overhead is a non-trivial matter, and is discussed in Section 4.1 on race conditions.

The use of these hash tables alone can result in suboptimal performance. While hash tables provide constant time lookup operation, there is a non-negligible performance overhead due to added lock contention, extra computation required to do the lookup, and some resulting cache pollution.

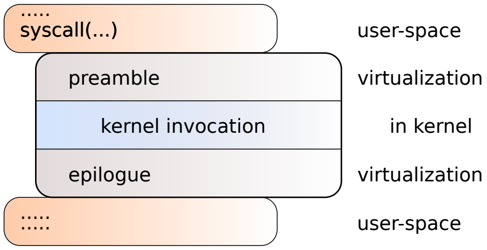

Figure 1: Anatomy of virtualization wrapper

In particular, the system-wide hash table is used for each resource access to determine the pod associated with the running process. The frequent use of this hash table can cause lock contention and impair scalability.

To minimize the cost of translating between pod namespaces and the underlying operating system namespace, we associate with each native process data structure a direct reference to the process's augmented virtualization state and the process's pod. These direct references act as a cache optimization that eliminates the need to use the table to access the virtualization data of a process, reducing the hash table lookup rate.

While this direct association only requires two references, it is unlikely that the native kernel process data structure has two unused references which can be used for this purpose. Instead, an effective solution is to extend the area occupied on the process's kernel stack by two pointers that reference the relevant data structures. In this manner, once a kernel process data structure is obtained, there is no need to refer back to any hash tables to translate from physical to virtual identifiers. Because this operation is so common, this reduces the virtualization overhead of the system across a broad range of virtualized system calls and eliminates a major potential source for lock contention.

For most of the subsystems referenced in Table 1, virtualization consists of managing the pod's state tables and handling the race conditions discussed in Section 4.1. This includes the following subsystems: PIDs, SysV IPC, network identifiers, and pseudo terminals, though the latter case of pseudo terminals requires further support through the filesystem. The filesystem, devices, and most of the miscellaneous systems are accessed through filesystem operations, whose virtualization specifics are discussed in Section 4.2. Unix IPC semantics are controlled using file descriptors with process groups, so our handling of fork in Section 4.1.2 is sufficient handling of Unix IPC virtualization. The remaining subsystem is pseudo filesystems which is discussed in Section 4.3.

4.1 Race Conditions

Race conditions occur due to the non-atomic transactions carried out by the wrapper subroutines. This is inherent to a virtualization approach that does not modify the kernel and treats kernel subroutines as black-boxes from the virtualization module's point of view. Race conditions, which make it difficult to maintain consistent virtual state, are more common in multiprocessor systems and preemptive kernels but are also present in non-preemptive uniprocessor systems.

Figure 1 illustrates the anatomy of a typical system call wrapper. Race conditions can either occur in the preamble or epilogue of the system call wrapper, and can also be distinguished as a race condition that is caused by identifier *initialization*, *deletion*, or *reuse*.

Preamble races can occur after a resource identifier has been translated from virtual to the corresponding physical identifier but before the underlying kernel code is invoked. The race occurs if, in this time frame, the resource is released in the kernel, and its physical identifier is subsequently reused and therefore ends up pointing to a distinct resource, possibly in another pod. The race conditions that are due to identifier reuse are generally rare given the very brief vulnerability window in which an unusual sequence of time consuming events must occur, and because the large size of the namespace from which offending identifiers are drawn. Allocation algorithms also attempt to avoid reusing a recently reclaimed identifier. Nevertheless, factors such as heavy workload, the presence of swapping activity, having large portions of the namespace already in use, and enabling concurrency can all contribute to the risk of a race event.

Epilogue races can occur after the kernel returns a physical identifier of a resource but before the virtualization wrapper converts it to a virtual identifier. Epilogue deletion races occur if the resource is freed during the period between kernel invocation and post-processing, which results in its removal from the virtualization state and causes the pending conversion to fail. Epilogue initialization races can appear between the time that a resource is allocated, which is usually after the completion of the underlying kernel code, and the time it is appropriately registered within the virtualization subsystem. Since these two operations are not executed within a single atomic section, the resource instance can be visible and modifiable to processes that do not belong to the same pod, or the resource instance can be exposed prematurely to processes in the same pod. In the following sections, we detail distinct problems and solutions and discuss the applicability of the patterns for other system resources.

4.1.1 Process ID Races

PID races can occur if a PID is referenced and changes during the execution of a virtualized system call such that stale data ends up being used. A change occurs when the PID is released and reclaimed by the kernel after a process terminates, and the PID may end up being reassigned by the kernel to a newborn process, as seen in Figure 2.

	Process A **pid=100** **vpid=400**	**Process B** **pid=110** **vpid=420**	**Process C**
1:	SYS_GETPGID(420)		
2:	virt_to_phys(110) ⇒110		
3:	kern_getpgid(110) ⇒110		
4:		SYS_EXIT(0) ⇒EXITED	
5:			CREATED ⇒ **pid=110** ⇒ **vpid=755**
6:	phys_to_virt(110) ⇒**755** ≠ 420		

Figure 2: PID deletion race. (1) Process A queries the PGID of process B, (2) we convert from virtual to physical, and (3) call the actual kernel system call. If (4) process B now exits, and (5) a new process C gains the same PID, then (6) we convert back from physical to virtual wrongly.

getppid and getpgid are examples of system calls vulnerable to these races. getppid begins with trivial preamble followed by invocation of the kernel's system call, and then translates the result from the physical value to the virtual one. An epilogue deletion race exists if the mapping of parent process's PID changes between the invocation and the translation, which can occur if the parent process terminates exactly then. The wrapper subroutine will fail if the PID is not reused, return an erroneous value if the PID is reused by a new process in the same pod, or return a meaningless value if the PID is reused by a new process in another pod. Similarly, getpgid begins with translating its PID argument from virtual to physical, then calls the kernel's system call, and wraps up by translating the return PID back from physical to virtual. It is exposed to the same epilogue deletion race as with getppid in addition to a preamble race.

Preamble races are potentially more harmful, especially for system calls that modify process state. For getpgid, a preamble race between the preamble and system call invocation can arise if the process terminates exactly then. One of four effects can occur as a result. First, if the PID is not reused, the kernel system call will return an error. Second, if the PID is reused and assigned to a new process in the same pod, the returned value will be the process group ID of another process in the same pod. Both of these cases are harmless as a similar race is inherent to Unix and may legally occur during its normal non-virtualized operation. Third, if the PID is reused and assigned to a process not in a pod, the physical process group ID returned by the kernel will not have a corresponding virtual group ID and the wrapper subroutine will fail. Fourth, if the PID is reused and assigned to a process in some other pod, the process group ID from another pod will be returned. This results in information leakage and violates the isolation between pods. It also causes inconsistency as two successive system calls will return different results. Moreover, other system calls that tamper with the system state can result in worse behavior. For instance, a race in the case of kill could end up delivering a signal from a process in one pod to another process is some other pod, and setpgid could attempt to modify a process group ID of a process in another pod, to a possibly undefined value there.

To prevent these races, we ensure that a reference count on the object in question is taken to guarantee that neither a PID nor the corresponding task structure are freed and reclaimed prematurely. This effectively protects the referenced object for the duration of the transaction. As long as that reference is held, the kernel will not reclaim the resource even if the process that owns it has exited. To implement this, we use the kernel's own reference count primitives for these objects and piggyback on them by calling the corresponding kernel subroutines to modify the reference count.

We minimize the interaction with the kernel by only modifying the kernel's reference count twice during the entire lifetime of a process. It is incremented when the process is associated with a pod, either by entering a pod or as a result of fork, and is decremented after the process exits. We combine this with a reference count that is maintained as part of the per process virtualization state. This count is initialized to one when the process joins a pod and modified twice in every transaction that is vulnerable to a PID race. It is incremented at the beginning of the transaction and decremented when the reference is no longer needed. The separation between the kernel's reference count on the original object and the module's reference count on the virtualized object reduces lock contention by preferring per pod locks over the kernel global lock. It also improves portability by reducing the dependency on the kernel without additional code complexity, since the reference count for virtualized objects is also required for other reasons.

	Parent **pid=100** **pid=400**	Child
1:	SYS_FORK()	
2:	kern_fork() ⇒ **pid=110**	
3:		CREATED ⇒ **pid=110**
4:		SYS_GETPID()
5:		kern_getpid() ⇒110
6:		phys_to_virt(110) ⇒**UNDEFINED**
	⇒ **vpid=420**	

Figure 3: PID initialization race. (1,2) The parent forks and (3) a child is created. The child executes before the parent completes the fork and (4) queries its PID. We (4) call the kernel system call, but (6) cannot convert back from physical to virtual because the virtual PID is uninitialized.

Similar to processes in a pod, regular processes not running in a pod are vulnerable to a symmetric race in which a regular process examines another regular process and the latter either enters a pod or dies before the former completes the transaction. If not addressed, this can result in an interaction between a regular process and a process in a pod, which should otherwise be forbidden. For example, consider a regular process that attempts to send a signal to another regular process. If the PID of the latter joins the scope of some pod, either by the owner entering or by being reused after the owner exits, after the sender already completed the PID translation but before it invoked the underlying native kill, the sender would end up delivering a signal to a process otherwise invisible to it.

We resolve this race by keeping a reference count for PIDs accessed outside of pods. Regular processes are associated with a special pseudo-pod. Referenced PIDs are then guaranteed not to be reclaimed prematurely. At the same time, processes are prohibited from entering a pod while a positive reference count exists for one of their PIDs. To complete the solution, we added the constraint that a process may only enter a pod executing in its own context. The rationale for this is similar to fork and other system calls; a fork cannot be imposed on a process, but rather must be executed by the process itself. This guarantees certain properties on the process state, such as well-defined state and a specific entry point which eliminate apriori numerous races and other subtleties.

4.1.2 PID Initialization Races

A key initialization race that must be addressed is correct initialization of the virtualization state of a process, particularly on process creation as a result of the fork system call, as seen in Figure 3. When fork is called, the virtualized system call performs some preliminary internal management and accounting, then invokes the native system call which returns twice, once in the context of the parent, and once in the context of the child. When executed in the context of the child process, the call returns immediately to user-level and does not return control to the kernel, so that no post-processing can be done as part of the virtualized system call execution by the child process. In particular, the child process is expected to return and begin executing before the parent process returns from the native system call.

As a consequence, there is a brief period in time in which the child process can resume execution without informing the virtualization module of its existence. Since the virtualization module is not aware that the child process had already been created, it is not able to initialize any necessary fields in its hash tables for that process, including any mappings between virtual and physical names for that child. Although the parent process can attempt to initialize the appropriate data structures on behalf of the child, it would only do so after its execution of the system call reaches the post-processing part. There is no guarantee for that to occur before the child process resumes execution in user-level, and potentially even issues other system calls. The problem is inherent to a system call interposition approach that treats `fork` as a black-box. As a result, it becomes difficult to determine whether the new child process belongs to a pod and must be virtualized. If the problem is not fixed, the process will not be isolated within a pod, and may freely interact with the underlying system and other processes.

In constructing an efficient method to ensure that a child process's state is properly initialized, a key observation is that an uninitialized process may execute freely as long as no interaction occurs with its virtualized state. As soon as such an interaction takes place, the process must first be initialized before it is allowed to continue execution. Assuming a method exists to detect that a process is not initialized, there are three cases to address. First, when a parent process returns from the native `fork` system call, it tests if the child process has already been initialized. If not, it initializes the virtualization state of the child, including storing the mapping of virtual and physical resource names in the appropriate virtualization data structure. Second, the nature of host-independent complete virtualization guarantees that the child process will not access any of its virtualization state until it calls a virtualized system call. As a result, the child process can wait until it calls a virtualized system call to have its virtualization state initialized. Each virtualized system call has a preprocessing step which tests whether the calling process is in a pod and whether its virtualization state has been initialized. When an uninitialized child process executes a virtualized system call, the system notices the uninitialized state and initializes the virtualization state at that time. Third, if some other process attempts to access the uninitialized child process via a virtualized system call, the child process is identified as being uninitialized which causes the system to initialize the virtualization state at that time. Conceptually the solution is to ensure that all direct and indirect access to the resource is virtualized, hence the first time the resource is accessed, it is also initialized.

To provide correct operation with low overhead, we augment the use of hash tables by storing some per process virtualization state as part of the in-kernel process data structure. The data structure used to represent a process in the kernel typically contains a set of flags used to note various process states. In general, the fields in the kernel process structure used to store such information are not completely populated so that unused parts remain. We use two bits of these unused parts to piggyback on the native `fork` system call to implicitly initialize a minimal virtualization state that identifies a process as being in a pod and uninitialized.

```
Syscall SYS_GETPID

if not in-pod then
    $pid \leftarrow kern\_getpid()$
    return pid
end if
if init-pending then
    call initialize_self()
end if
$pod \leftarrow lookup\_pod()$
$pid \leftarrow kern\_getpid()$
$vpid \leftarrow phys\_to\_virt(pid)$
return vpid
```

Figure 4: Pseudo code for `getpid` wrapper

These bits serve as two helper flags for virtualization. The first is the *in-pod* flag and indicates whether a process is in a pod. A crucial advantage of using this field is that it is inherited across child process creation given the semantics of `fork`. A child of a process that is already in a pod atomically inherits the flags and is therefore immediately identified as also being in a pod. Thus, processes in pods can be readily and efficiently filtered and made invisible to regular processes, even when uninitialized. The second flag is the *init-pending* flag and indicates that a process in a pod is pending initialization. A parent process sets its *init-pending* flag when processing the virtualized `fork` system call before executing the native system call. The flag is inherited atomically by the child process so that both the parent and the child process appear to be uninitialized. The presence of the flag on the parent is meaningless and ignored. The flag is cleared on both when the child process has been initialized.

The combined approach of employing the helper flags in addition to the hash tables provides a performance benefit as well. Hash table lookup is part of the critical path of four common tasks: first, when testing whether a process belongs to a pod when it issues a system call to decide whether to virtualize or not; second, when testing whether a target process (e.g. in a pod) should be masked out from another process (e.g. not in a pod); third, when access to the virtualization data of a process is needed; and fourth, when a physical-to-virtual translation or vice versa is required. While hash tables provide constant time lookup operation, there is a non-negligible performance overhead due to added lock contention, extra computation required to do the lookup, and some resulting cache pollution.

Testing for the flag on a process trivializes the first two tasks and eliminates the need to perform a hash table lookup. This is particularly beneficial for regular processes not running in pods that would otherwise suffer a performance degradation due to such a lookup in each virtualized system call. For example in Figure 4, a negative return value on the *in-pod* check at the beginning of the wrapper short-circuits the `lookup_pod()` call. Since PIDs-related functions are a main part of the code path, the overhead for the system as a whole is lowered. Isolation of processes in a pod from processes outside of the pod becomes easy as well: if a process not running in any pod attempts to access a process in a pod, the *in-pod* flag of the process in a pod will already be set and hence it is straightforward to deny access.

4.1.3 SysV IPC Races

SysV IPC [31] primitives consist of message queues, shared memory and semaphores. Unique identifiers refer to active instances. Keys are used to obtain identifiers. IPC identifiers and keys are global resources that must be virtualized using virtual-to-physical and physical-to-virtual hash tables. They are also mutable during system calls, potentially resulting in initialization, preamble, and epilogue deletion races. For simplicity, we focus our discussion on message queues, but the same principles apply for the other two primitives.

Initialization races can occur when a new identifier is created as it may become visible to the system prior to the initialization of its virtualization data. For instance, an IPC identifier allocated inside a pod whose virtualization state has not yet been initialized cannot be determined to belong to the pod, and therefore may be potentially accessed by a process outside the pod. This issue is aggravated since most IPC primitives will alter the system state, possibly before the resource is ready. Similar to the solution to the process initialization race condition we are assured that the resource cannot be accessed without going through the virtualization layer. Unlike processes however, the internal data structure that represents IPC resources is not extensible, making it impossible to associate either a pointer or a flag with it.

To prevent misuse of IPC identifiers before their virtualization data is initialized, we introduce a third hash table called the *outside-pod* table to indicate whether a given instance has been initialized. The outside-pod table stores all identifiers in use by processes not in a pod. As with PIDs, this may be thought of as treating the namespace that does not belong to any pod as a pseudo-pod where virtual and physical identifiers are mapped one-to-one. Regular processes must consult the outside-pod table to access an identifier, analogous to testing the *in-pod* flag for PIDs. They will be blocked from accessing uninitialized identifiers since they will fail to find them in the outside-pod table.

The outside-pod table must be correctly populated to account for IPC resources that may already exist when the virtualization module is loaded. When the module is loaded, it must scan the kernel data structures for instances of IPC primitives and place the identifiers in the outside-pod table. Special care must be taken not to overlook an instance that is being created at the time of the scan or afterwards, by a process that started a native (non-virtualized) system call prior to the scan. Otherwise, that identifier will not be accounted for and will consequently become invisible to all processes, including the process that created it. A performance issue with this scheme is the added overhead to IPC related system calls for processes that do not belong to a pod as every operation on an identifier by a process not in a pod implies a lookup in the outside-pod table.

Preamble races can occur due to identifier reuse, as seen in Figure 5. For example, consider a process in a pod that holds a valid message queue identifier and calls `msgsnd` to send a message. Suppose that after the translation from the virtual namespace to the physical namespace by the wrapper subroutine, another process in the same pod deletes that message queue from the system, then subsequently that identifier is reused for a new message queue in another pod. When the first process invokes the native system call, it will end up violating the isolation semantics between pods. Since the semantics of IPC allow to remove instances at any time regardless of how many processes may be using it, the kernel does not keep a usage count on them, thus hindering the piggybacking on a native reference count to handle said races similarly to PIDs.

	Pod 1: **ipc id=10** **ipc vid=55**		**Pod 2:**
	Process A	**Process B**	**Process C**
1:	SYS_MSGSND(55, ...)		
2:	virt_to_phys(55) ⇒10		
3:		SYS_MSGCTL (55, IPCRM,...)	
4:		virt_to_phys(55) ⇒10	
5:		kern_ipcrm(10) ⇒deleted	
6:			SYS_MSGGET(...) ⇒10
7:	kern_msgsnd(10, ...) ⇒**ILLEGAL**		

Figure 5: IPC reuse race. (1) Process A sends to queue with virtual ID 55. (2) we convert the ID from physical to virtual. (3,4,5) process B deletes the queue in the same pod, and (6) process C allocates a new queue with the same ID in another pod. When (7) process A calls the actual kernel system call, it sends the message illegally across pod boundary.

To prevent improper reuse of identifiers we propose a fourth hash table named *restricted-ID* table to track all the instances which are being referenced to at any time by the virtualization code together with their reference count. Identifiers will be inserted to the table during the preamble, and taken out if the reference count drops to zero in the epilogue. The virtualization code of `msgget` will inhibit reuse of an identifier as long as it appears in the table, by having the epilogue inspect new identifiers to ensure that they are not restricted. If they are, the epilogue will deallocate that instance and a new allocation is attempted.

However, this scheme is still incomplete as illustrated by the following subtlety: suppose a process calls `msgsnd` with some identifier and is preempted between the preamble and the invocation of the actual system call. Suppose also that another process (in the same pod) now removes that instance, and a third process in another pod allocates a new message queue with the same identifier, but is preempted before testing it against the restricted-ID table. If the original process now kicks in it will eventually access the new message queue rather than the intended one. The outcome clearly undermines isolation between pods.

A simple solution is to have the epilogue of `msgget` mark the virtualization state of identifiers that were deallocated so the epilogue of `msgsnd` can detect this condition. When this occurs, `msgsnd` responds by retrying the operation. This is sufficient to ensure that the pod boundaries are respected, since the offending (newly allocated) identifiers never gets a chance to be used in the other pod. Once interaction is confined to the original pod, any side effects are legal since similar circumstances can occur on the native OS.

A performance issue with the above is the added overhead to IPC related system calls, even for processes that run outside any pod. This overhead comes directly from the extra bookkeeping by the virtualization logic. First, every creation of an instance, either inside or outside a pod, involves a lookup in the restricted-ID table. Second, every operation on an identifier requires that the identifier be inserted in the restricted-ID table for the duration of the system call. Finally, every operation on an identifier by a process not in a pod implies a lookup in the outside-pod table.

Unlike PIDs, IPC consists of two inter-related resources, namely identifiers and keys. Both are global and not associated with specific processes that own a reference to them. Keys identify a context and are persistent while identifiers are created when such contexts are instantiated to allocate a message queue, hence representing a specific instance. Once a key has been instantiated, future attempts to instantiate it will resort to the existing instance, until that instance is explicitly deleted. For example, the first call to `msgget` with some key value will allocate a new message queue and assign a unique identifier that represents that key. Subsequent calls will detect the active queue that is associated with the specified key and will return the same identifier, until finally the queue is removed, and so on.

IPC keys are unusual in that the user can select the value of a key when allocating a new message queue. With all other kernel resources, their physical names are assigned solely by the kernel. Since the kernel always selects unique identifiers, it is not possible for two distinct resources to have the same physical name. In contrast, values of keys are set forth by the application and may potentially coincide across two distinct pods. If the same key is used in two separate pods and passed to the OS for allocating new message queues, the OS would not create a message queue in each pod. Instead, it would incorrectly create a single queue and provide the same queue identifier in both pods.

To address this problem, we leverage a special key value, `IPC_PRIVATE`, designed to allocate private message queues that are not associated with any specific key, and whose identifier cannot be obtained by a subsequent call to `msgget`. The virtualization wrapper of `msgget` first searches for the given virtual key in the corresponding hash table and returns the corresponding virtual identifier if found. Otherwise, it invokes the original system call, substituting the original key argument with the special `IPC_PRIVATE`, causing the OS to generate a private queue. By creating private message queues, we ensure the uniqueness of each queue within the system. After the system call returns a new identifier, the epilogue allocates a corresponding virtual identifier, populates the table with a new mapping, and associates the virtual key with the virtual ID.

The special behavior of IPC allocation and its virtualization leads to a unique type of preamble race condition. Consider two processes in the same pod trying to allocate a message queue with the same key. Under normal circumstances, the one that is scheduled first will receive the identifier of a new queue, and the other will receive the same identifier. However, if both processes complete their preamble before either of them invokes the real system call, the preamble will have replaced the original key with `IPC_PRIVATE` and they will now each obtain a distinct, private queue.

The kernel already serializes certain types of IPC calls, like creation, deletion and manipulation of message queues–but not their actual use–with semaphores that ensure mutual exclusive modifications. Since the offending system calls are already serialized by design, we can use a matching semaphore to protect the virtualization wrapper and make the entire virtualized operation atomic without compromising scalability. This solution also eliminates other race conditions such as epilogue deletion races. A deletion race can only occur when a physical-to-virtual translation of IPC identifiers takes place. In the IPC context this translation only happens during allocation. However, deletion and allocation are mutually exclusive by use of the semaphore and are therefore protected from this race.

4.1.4 Pseudo Terminals Races

Pseudo terminals [31] (PTS) are pairs of master/slave devices whose input and output streams are cross linked. The slave end emulates the behavior of a line terminal for the process using it. When the master PTS multiplexer device is opened, a corresponding inode for a slave device is created in the `devpts` pseudo filesystem and named `/dev/ptsN`, where N is the device minor number. The inode is destroyed when the device is released. In addition to virtualizing the pseudo terminal name (the device minor number), it is essential to virtualize the entries in `/dev/ptsN` to export adequate views in the contexts of different pods. It also prevents races arising from having indirect paths to the resource. This is discussed further in Section 4.3.

The only conceivable system call involving pseudo terminals is for a process to query the identifier of a terminal attached to a file descriptor. This is always race-free since the initialization of the pseudo terminal must have already been completed when the matching `open` system call terminated, and the deletion of the pseudo terminal could not have occurred since it would have required that the terminal be closed, yet the said file descriptor is still kept open. While pseudo terminals are not subject to preamble and deletion races, they are subject to initialization races. The initialization pitfall is similar to the problem with IPC where a process not in a pod may be able to access a recently created slave device in a pod prior to the completion of its setup. A plausible approach is to use an identical solution to the outside-pod table used with IPC to completely mask out pseudo terminals while not yet initialized.

We employ a lower overhead approach that takes advantage of the ability to borrow one bit from the in-kernel data structure of pseudo terminals. We use the bit to store a per device *init-complete* flag that marks the device as initialized when it is set. Since the bit is unused, the flag is not set upon creation of a pseudo terminal. A newly created pseudo terminal without the flag set is noted as uninitialized and is made invisible to any process trying to access them, be it inside or outside a pod. The flag is set once the initialization is successfully completed and the appropriate associations in the translation hash tables have been made, making the pseudo terminal finally accessible within its pod, or among regular processes if it was originally created outside of any pod. This approach is similar to the *init-pending* flag used for dealing with process initialization races. Using a per-instance flag instead of a global hash table improves scalability and performance by avoiding the need to serialize access to a table, and by eliminating both the extra cycles of the lookup and the associated cache pollution.

Similar to the outside-pod table with SysV IPC, it is possible that access to a pseudo terminal not belonging to any pod is denied for a brief period from processes not in a pod that should otherwise be able to access it. This can happen in the period between the creation of the new inode and the final completion of its initialization. Since such behavior can be expected in traditional Unix, we regard this as a non-virtualization issue. When the virtualization module is loaded, it is necessary to scan the kernel for already open pseudo terminals to set their initialization flags. This must be done carefully to account for rare races that can occur during the scan itself.

4.2 Filesystem Virtualization

To provide modular support for multiple filesystems, many commodity OSs provide a virtual filesystem framework that supports a form of interposition known as filesystem stacking [38]. We leverage this support along with the `chroot` utility to simplify filesystem virtualization. Filesystem virtualization is accomplished by creating a special directory per pod that serves as a staging area for the pod's private filesystem hierarchy. Storage requirements are minimized by sharing read-only portions of the filesystem among pods, if applicable, through loopback mounting or networked filesystems. The `chroot` system call is used to confine processes that belong to the pod within their private subtree. To ensure that the root of that filesystem is never traversed, we use a simple form of filesystem stacking that overloads the underlying filesystem permission function to implement a barrier directory that enforces the `chroot`-ed environment and ensures that it is only accessible to files within the owner pod. This use of filesystem stacking leverages existing kernel functionality and avoids the need to replicate that functionality as part of the virtualization implementation.

4.3 Pseudo Filesystems

Pseudo filesystems are memory-based filesystems that provide the user with an interface to kernel resources and facilities. Pseudo filesystems share three key properties. First, they provide a public, indirect path to a view of global resources. Second, creation and deletion of resource instances are reflected dynamically in this view because the actual behavior of the resources is tracked using dedicated callback subroutines. Third, they may generate a process specific view that is context dependent and differs among processes (e.g. the symbolic link `/proc/self`). To ensure proper virtualization semantics, we must virtualize these views to provide context dependent views corresponding to the respective pod being used. We briefly discuss how this is done cleanly and simply for two important pseudo filesystems, `devpts` and `procfs`.

The `devpts` filesystem provides an interface to pseudo terminals. Similar to the filesystem virtualization described earlier, we use a filesystem stacking approach to virtualize `devpts`. Given that each pod uses a dedicated subtree of the filesystem as its root filesystem, we provide a pod `devpts` by stacking an instance of a virtualization filesystem on the `/dev/pts` directory in each pod. This is a very lightweight stackable filesystem whose sole purpose is to virtualize the underlying filesystem in the context of the specific pod in which it resides. In addition, the required logic is completely independent of the specific pseudo filesystem, which significantly reduces complexity while leveraging the generality and portability of filesystem stacking.

The `procfs` filesystem maintains a view of the processes running in the system as well as of system properties. A key feature is that it provides an exported interface that loadable kernel modules can use to dynamically extend its layout. We harness this dynamic extensibility to provide each pod with the requisite context dependent view. For each pod, the virtualization layer automatically creates a private subtree within the `procfs` hierarchy by mirroring the original filesystem structure. To keep the overhead low, we do not replicate code or create additional inodes, but instead use hardlinks to refer to existing inodes. This subtree is loopback-mounted at the appropriate point (`/proc`) within the pod's root subtree. This approach is appealing due to the simplicity and lowered virtualization overhead compared to other approaches such as filesystem stacking.

5. EXPERIMENTAL RESULTS

To demonstrate the effectiveness of our OS virtualization implementation, we measured its performance on both micro-benchmarks and real application workloads. We quantify the overhead of OS virtualization, measure the additional cost of correctly handling various race conditions, and also present measurements using hardware virtualization to provide a basis of comparison between virtualization approaches.

We ran our experiments on both Linux 2.4 and 2.6 kernels, and on both UP and SMP systems. However, we only present data for Linux 2.6 on SMP due to space constraints. The operating system configuration used was Debian Stable. We conducted the measurements on an IBM HS20 eServer BladeCenter. Each blade had dual 3.06 GHz Intel Xeon^{TM} CPUs, 2.5 GB RAM, a 40 GB local disk, and was connected to a Gigabit Ethernet switch. Hyperthreading was enabled for the SMP kernel measurements.

We performed our experiments on three configurations: *Base*—a vanilla Linux system providing a baseline measure of system performance; *With*—*Base* with our virtualization module loaded but no pods instantiated, providing a measure of the overhead incurred by regular processes running outside of a pod; *Pod*—*Base* with our virtualization module loaded and benchmarks executed inside of a pod, providing a measure of the overhead incurred when running inside of a pod.

5.1 Micro-benchmarks

To measure the basic cost of OS virtualization, we used micro-benchmarks to measure the performance of different system calls with different types of virtualization overhead. Seven micro-benchmarks were used. Each ran a system call in a loop and measured its average execution time: `getpid`, `getsid`, `getpgid`, `fork`, `execve`, `shmget`, and `shmat`. In particular, the `fork` benchmark forked and waited for a child that exited immediately, the `execve` benchmark forked and waited for a child that executed a program that exits immediately, the `shmget` benchmark created and removed IPC shared memory segments, and the `shmat` benchmark attached a segment of shared memory twice, modified both copies, and detached it.

Figure 6 shows the execution time of each benchmark on each configuration using a SMP kernel and normalized to *Base*. Each bar shows the actual execution time reported by the benchmark. Comparing *With* and *Pod* with *Base*, the measurements show that operating system virtualization overhead is quite small in all cases, generally 2-4% and less than 35% in the worst case.

We performed a simple comparison on the UP kernel to quantify the cost of using `ptrace` for system call interposition in lieu of our kernel module implementation. We executed a version of the `getpid` micro-benchmark that consists

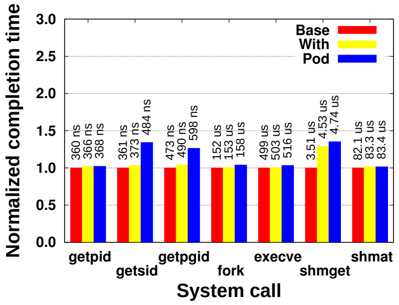

Figure 6: Virtualization cost (micro)

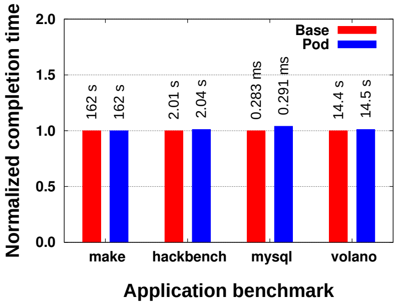

Figure 7: Virtualization cost (macro)

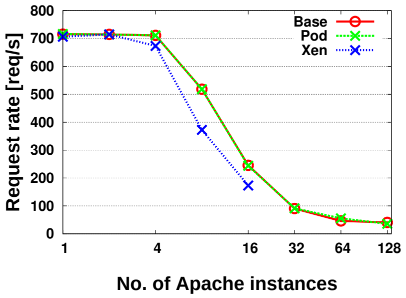

Figure 8: Virtualization scalability

of two processes: a tracer and a tracee. The tracer process is notified about every entry and exit of system calls of the tracee. It then peeks into the tracee's memory, emulating the work of the wrapper's preamble and epilogue. The tracee executes `getpid` repeatedly and we measured the average execution time for the system call. The average execution time was 5.5 μs for tracing without peeking and 7.7 μs for tracing and peeking. `getpid` degrades by a factor of 13 just for monitoring, and by a factor of 20 if the tracer also peeks into the tracee's memory. This overhead does not even take into account the added cost of basic virtualization functionality. In comparison, our kernel module virtualization overhead is only 2 percent for the same system call.

We also compared our measurements with results reported for in-kernel interposition mechanisms. SLIC [13] reports 10% overhead due to the basic dispatcher code, roughly 35% for the interception of `getpid`, and somewhat lower for more involved system calls. However their base system was a slower UltraSparc and we expect their overhead would be much less on a more modern system. Systrace [28] reports 30% overhead for the same case, which includes their security policy checks, on hardware that is similar to ours. Our measurements suggest that a loadable kernel module implementation is not outperformed by an implementation that modifies the kernel directly.

The actual *With* and *Pod* execution times for the first three benchmarks shown in Figure 6 provide a quantitative measure of the basic cost of OS virtualization functionality. `getpid` only adds a test of the *in-pod* and *init-pending* flags to determine whether the process is in a pod, and if so, whether it is pending initialization. The overhead is 8 ns and represents the minimum overhead of a virtualized system call. *With* and *Pod* execution times are the same for `getpid` because the cost of obtaining the virtual identifier is negligible since it is stored as part of the per process virtualization state, which is directly referenced; no hash table translation is required. Compared to `getpid`, `getsid` also uses the hash table to translates the return value from physical to virtual identifier if a process is in a pod. This translation overhead is 111 ns as indicated by the difference between *With* and *Pod* `getsid` execution times. Most of this time is due to the extra locking mechanisms required for correct operation on SMP. Consequently, the overhead is noticeably lower for UP, where the same translation takes approximately 12 ns. `getpgid` also needs to do a hash table lookup and modify the kernel's process reference count for the *With* case, as discussed in Section 4.1.1. This added 17 ns versus the *Base* case due to the lookup, since the reference count adds negligible overhead. For the *Pod* case, `getpgid` also needs to modify the virtualization module's process reference count and perform an additional hash table translation, adding 108 ns versus the *With* case, mostly due to locking mechanisms.

While more complex system calls require more OS virtualization logic, the overhead of the additional logic is amortized by the additional overhead of the native system call. For example, even though the virtualization cost of `fork` is roughly 6 μs as shown in Figure 6, the virtualization overhead as a percentage of the system call execution time is only 4%. The overhead is due to allocating and preparing the virtualization data structure for the child process, linked list maintenance and ensuring correct initialization. All of the more complex system calls have small overhead except for `shmget`, which has 35% overhead for *With* and *Pod* compared to *Base*. The higher overhead here is largely due to the use of an additional semaphore as discussed in Section 4.1.3, which does not compromise scalability, but it does increase execution time.

Most of the micro-benchmarks took shorter to run on an UP system versus a SMP system, due to the kernel's locking mechanisms, which are trivialized in the UP kernel. On a UP system, OS virtualization overhead was noticeably lower for three of the micro-benchmarks, `getsid`, `getpgid`, and `shmget`. In those cases, the extra locking mechanisms result in more overhead for processes running in a pod. The cost of these mechanisms is more prominent for the simple system calls but is amortized for the more complex system calls. However, the cost of synchronization is noticeable for `shmget` because it requires the use of a semaphore which is much more expensive than simple spin locks. Except for `getsid`, `getpgid`, and `shmget`, OS virtualization overhead was roughly 3 percent or less in all other cases.

5.2 Application Benchmarks

To provide a more realistic measure of virtualization cost that is expected in actual use, we measured the performance of different virtualization approaches using five different application workloads: *make*—complete build of the Linux kernel using `gcc` 3.3 (`make -j 10`); *hackbench*—a scheduler performance scalability benchmark which creates many processes in groups of readers and writers sending small messages [15] (32 groups); *mysql*—Super Smack 1.3 database benchmark using MySQL 4.1.9; *volano*—VolanoMark Java chat server benchmark 2.5.0.9 using Blackdown Java 2 Runtime Environment 1.4.2-02; *httperf*—httperf 0.8 web server performance benchmark using Apache web server 2.0.53.

Figure 7 shows the execution time of the make, hackbench, mysql, and volano benchmarks on *Base* and *Pod* using an SMP kernel with all measurements normalized to *Base*. (We omit the *With* case, since the micro-benchmarks results already show that the overhead for regular processes running outside a pod is negligible.) Each bar shows the actual execution time reported by the benchmark. The mysql benchmark reports queries/s, but is converted to query service time to be consistent with the other benchmarks. The measurements show that *Pod* provides comparable performance to *Base*. OS virtualization overhead as shown by the *Pod* case is 2% or less for all applications.

To provide a measure of performance scalability, we measured the performance of httperf as we scaled the number of instances of the benchmark running at the same time. For *Base*, we ran multiple instances of Apache with each instance listening on a different port. For *Pod*, we ran an Apache instance in each pod. We scaled the number of httperf and Apache instances from 1 to 128 and measured the average request rate across all instances.

We also executed this benchmark inside a Xen virtual machine running a Linux OS, to provide a measure of the overhead of hardware virtualization. Xen was used given its claims of superior performance versus other hardware virtualization systems [3]. For Xen, we ran an Apache instance in each Xen VM. To enable Xen to scale to a larger number of Apache instances, we configured the Xen VM used with 128 MB RAM.

Figure 8 shows the results of this experiment. As expected, the average httperf performance per instance decreases as the number of instances increases due to competition for a fixed set of hardware resources. httperf performed similarly on all systems when only one instance was executed. However, as the number of instances increased, httperf performance on Xen falls off substantially compared to its performance on *Base* or *Pod*. Xen scalability was further limited by its inability to run more than 16 application instances at the same time because it could not allocate any additional VM instances given the 128 MB RAM per VM and the 2 GB RAM available in the machine used. In contrast, both *Base* and *Pod* continue to provide comparable scalable performance up to 128 instances. It is worth noting that in addition to performance scalability, Xen is also limited by storage scalability since each VM requires a separate OS image. Since OS virtualization does not require separate OS images per virtual execution environment, it does not suffer from this storage limitation.

6. RELATED WORK

Given its various benefits, a number of OS virtualization systems have been implemented over the past decade. Capsules [30] was an early effort in Sun's Solaris operating system. This research inspired Zap [24], the earliest system to provide complete, host-independent virtualization in a kernel module without base kernel changes. Our work builds on previous work on Zap, but discusses for the first time various key implementation issues that arise in practice in efficiently supporting complete, host-independent OS virtualization while preserving scalable OS performance. Some virtualization systems that do not provide complete, host-independent virtualization have been implemented in Windows [9, 37] without kernel changes, in part by interposing at the DLL level with the associated disadvantages of user-level interposition described in Section 2. Other recent virtualization systems have been implemented by making extensive kernel changes to restructure the underlying operating system [27, 32, 34].

Building on ideas from Capsules, Zap, and other early virtualization systems, some commodity OSs are gradually incorporating virtualization support by making pervasive changes to the OS kernel. The implementation of per-process namespaces in the Linux kernel [5, 22] began in 2005 and is still in progress. This effort has affected nearly every corner of the Linux kernel. In contrast, our approach is minimally invasive since it treats the OS kernel as an unmodified black box. This makes it easier to backport to fit commodity OS kernels, and the use of a kernel module simplifies deployment.

Interposition has been the subject of extensive research, and is repeatedly relied on by a host of approaches for security and other general OS extensions, spanning all three approaches: user-level methods [1, 2, 14, 17, 18, 20] and [35] (the latter presenting an excellent analysis of `ptrace`), kernel modifications [4, 8, 16, 28, 34], loadable kernel modules [10, 11, 13] and hybrid methods [12]. Some of these have pointed out severe limitation of the security of user-level based interposition mainly due to vulnerability to races as well as bypassability. Our work reiterates these deficiencies in the context of virtualization. We have not found any work that deals with the issue of referencing unexported kernel functionality or data structures. Garfinkel [11] focuses on process isolation in the context of security whereas our work pertains to OS virtualization, which not only isolates but also decouples the process from the OS instance. Virtualization addresses a superset of the race conditions associated with isolation, introduces new classes of races such as initialization and deletion races, and is required to keep references on OS resources. Our approach is also concerned with performance considerations in addressing these races, and in particular scalability.

7. CONCLUSIONS

While OS virtualization concepts have been previously discussed, previous work does not address important implementation issues in supporting OS virtualization in the context of commodity OSs. To the best of our knowledge, our work explores these implementation issues in depth for the first time. We discuss the need for system call interposition for implementing OS virtualization and compare various approaches for providing this functionality. We demonstrate the benefits of a loadable kernel module implementation and show that the overhead of this approach is substantially less than other approaches such as using process tracing functionality. We discuss how OS virtualization state should be stored and describe several important optimizations for ensuring low performance overhead. Furthermore, we discuss in detail various race conditions that can arise in implementing OS virtualization and how these race conditions can be addressed. We built an OS virtualization prototype as a Linux kernel module that works across multiple kernel versions, demonstrating the portability of our approach. Our experimental results demonstrate that OS virtualization can be achieved with very low overhead.

8. ACKNOWLEDGEMENTS

Dan Phung provided helpful comments and edits on earlier drafts of this paper. This work was supported in part by NSF grants CNS-0717544, CNS-0914845, and CNS-0905246, and AFOSR MURI grant FA9550-07-1-0527.

9. REFERENCES

[1] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Applications. In *Proceedings of the 2000 USENIX Security Symposium*, Denver, CO, Aug. 2000.

[2] R. M. Balzer and N. M. Goldman. Mediating Connectors: A Non-Bypassable Process Wrapping Technology. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Austin, TX, June 1999.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, and I. P. A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[4] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of 1995 USENIX Winter Conference*, New Orleans, LA, Jan. 1995.

[5] S. Bhattiprolu, E. W. Biederman, S. Hallyn, and D. Lezcano. Virtual Servers and Checkpoint/Restart in Mainstream Linux. *SIGOPS Operating Systems Review*, 42(5), July 2008.

[6] T. Boyd and P. Dasgupta. Process Migration: A Generalized Approach using a Virtualizing Operating System. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[7] How to Break Out of a Chroot Jail. `http://www.bpfh.net/simes/computing/chroot-break.html`.

[8] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th USENIX Systems Administration Conference (LISA)*, New Orleans, LA, Mar. 2000.

[9] P. Dasgupta and T. Boyd. Virtualizing Operating System for Seamless Disributed Environments. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Nov. 2000.

[10] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[11] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, San Diego, CA, Feb. 2003.

[12] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the Network and Distributed Systems Security Symposium*, San Diego, CA, Feb. 2004.

[13] D. P. Ghormley, D. Petrou, S. H. Rodriguez, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, Berkeley, CA, June 1998.

[14] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 1996 USENIX Annual Technical Conferemce*, San Jose, CA, July 1996.

[15] Hackbench. `http://developer.osdl.org/craiger/hackbench`.

[16] S. Ioannidis and S. Bellovin. Sub-Operating Systems: A New Approach to Application Security. Technical Report MS-CIS-01-06, University of Pennsylvania, Feb. 2000.

[17] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, Feb. 2000.

[18] M. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, Dec. 1993.

[19] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International SANE Conference*, Maastricht, The Netherlands, May 2000.

[20] E. Krell and B. Krishnamurthy. COLA: Customized Overlaying. In *Proceedings of 1992 USENIX Winter Conference*, San Francisco, CA, Jan. 1992.

[21] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[22] PID Namespaces in the 2.6.24 Kernel. `http://lwn.net/Articles/259217/`.

[23] M. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[25] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, 1974.

[26] S. Potter and J. Nieh. Reducing Downtime Due to System Maintenance and Upgrades. In *Proceedings of the 19th Large Installation System Administration Conference (LISA)*, San Diego, CA, Dec. 2005.

[27] D. Price and A. Tucker. Solaris Zones: Operating Systems Support for Consolidating Commercial Workloads. In *Proceedings of the 18th Large Installation System Administration Conference (LISA)*, Atlanta, GA, Nov. 2004.

[28] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[29] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[30] B. K. Schmidt. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, CS Department, Stanford University, 2000.

[31] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison-Wesley, 1993.

[32] Parallels Virtuozzo Containers. `http://www.parallels.com/products/virtuozzo`.

[33] VMware, Inc. `http://www.vmware.com`.

[34] Linux VServer Project. `http://www.linux-vserver.org`.

[35] D. Wagner. Janus: An Approach for Confinement of Untrusted Applications. Master's thesis, University of California, Berkeley, Aug. 1999.

[36] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[37] Microsoft Application Virtualization. `http://www.microsoft.com/systemcenter/appv/default.mspx`.

[38] E. Zadok. *FiST: A System for Stackable File System Code Generation*. PhD thesis, Computer Science Department, Columbia University, May 2001.