Ultralight Kernel Virtualization with Runnable Userspace Meta Programs

Antti Kantee
Helsinki University of Technology
pooka@cs.hut.fi

Abstract

The common approach for virtualization is to treat the operating system as the basic unit. This gives the benefit of transparency, but is a heavyweight approach. We have implemented a radically different approach: virtualize based on application requirements and include only necessary components, e.g. networking domains. The virtual machine monitor is provided by a standard unmodified operating system.

As opposed to implementing a new operating system, we have modified NetBSD and show that an existing Unix-style OS can be virtualized and componentized non-intrusively with relative ease; our changes are a part of the upstream source tree. We support file systems, networking, and a selection of drivers. Furthermore, we are able to use standard NetBSD binary kernel modules under virtualization.

The bootstrap of a new virtual kernel takes milliseconds in the typical case. Resource consumption depends on the code base being virtualized and for NetBSD an untuned base virtual image uses 300kB of memory and e.g. a fully virtualized TCP/IP stack instance total is 500kB. This means scalability to thousands of units on commodity server hardware. Performance as compared to a regular kernel depends on the workload and ranges from superior to similar to worse.

Categories and Subject Descriptors D.2.13 [*Reusable Software*]: Reusable libs; D.4.5 [*Reliability*]: Fault-tolerance; D.4.7 [*Organization and Design*]: Interactive systems

General Terms Design, Measurement, Performance

Keywords Virtualization, Lightweight, Kernel service

1. Introduction

Virtualization can be used to provide multiple instances of operating systems on a single unit of hardware. This is useful for example for fine tuning each instance for an individual application, providing sandboxes where users can be given administrator privileges, and kernel code development.

Virtualization is very widely used in the real world [1, 2, 9]. The likely reason for its success is the ability to virtualize operating systems which were in use already. The resulting extended and virtualized environment looks and feels similar to a non-virtual one. In case the entire entire operating system is virtualized, there is no difference to a non-virtual one from a practical standpoint.

However, full virtualization does also virtualize the aspects of the operating system that targeted cases are not interested in. For example, an application requiring TCP/IP stack virtualization for simulating a massive routed network with thousands of virtual nodes is not interested in file system support or most likely even the tty driver.

Our approach is the opposite of full virtualization: we include only what is the bare necessity for our target system by breaking the kernel into four independent coarse-grained components: base kernel, networking, file systems, and device drivers. Upon these basic components various configurations can be built, including applications which use the virtualized components and drivers. This approach complements existing virtualization technologies instead of competing with them.

We call our model *Runnable Userspace Meta Programs*, or rump for short. The name is due to the fact that we use userspace processes for virtualization containers. A kernel service running in userspace is called a *rump kernel*.

Our goal not to solve all problems or provide a general solution. Our goal is provide virtualization of kernel services with minimal overhead. An example usecase is consumer embedded devices. Manufacturers desire a lightweight method for segregating users from system services. For example, the TCP/IP networking service used by user-installed applications should be different from the TCP/IP networking service used by the system internally. This enables better control of system component behavior. Since resources are at a premium in embedded systems, full virtualization is out of the question.

Rump does not require a special virtual machine monitor. This means that we do not have to run another layer of processes, threads, context switching, synchronization primitives, etc. on top of the process facility in userspace, but rather use functionality readily supplied by the host operating system. After all, the purpose of a process is to provide the illusion of only one application running on the system – it is a virtualization container waiting to happen. Leaving out extra layers also allows us to avoid the slowdown of virtual operating systems running on top of operating systems [21].

The idea of running kernel code directly in a userspace process is not new. We have used it successfully ourselves for file systems [20]. The work in this paper explores the concept much further: it expands the idea to networking and device drivers, offers a system call interface to the rump kernel with 111 supported syscalls and supports RPC for an "overlay virtual OS" consisting of multiple processes. Also, to the best of our knowledge, rump is the only paravirtualization facility which can use binary kernel modules from the OS being virtualized. Finally, we formulate and provide OS requirements for rump virtualization.

We support two classes of device drivers: so-called pseudo devices, which are not backed by hardware, and USB devices. Support for the latter is in the beginning stages, but we can already access file systems on USB sticks by running the kernel driver stack in userspace (cf. Figure 1).

The implementation is done on an existing open source Unix-based operating system, NetBSD. We aim for a pragmatic approach and our goal is a working and usable operating system both in a virtualized and non-virtualized context. After its initial import, rump has been developed in the NetBSD main source tree for over 2 years now. This stands as a testament to the fact that the approach is applicable in a real-world operating system.

The rest of the paper is structured as follows. Section 2 takes a closer look at virtualization technologies available today and defines our approach. Section 3 explains the implementation in detail. Section 4 evaluates the technology verbally and Section 5 provides measurement data. In Section 6 we formulate the requirements for an OS to be subjected to rump virtualization and Section 7 presents applications. Finally, Section 8 provides concluding remarks.

2. Virtualization

We use the following terminology in the discussion: a *virtual machine monitor* is the software layer which provides virtualization. The *host operating system* or *host* is the entity running the virtual machine monitor. The *guest* is the virtualized operating system.

There are three main types of operating system virtualization available today:

- **instruction set virtualization** [2, 12]: the virtual machine monitor interprets the instructions made by the guest and translates them to a form suitable for the host. This can be either runtime translation or pre-translation and for the full instruction set or just the privileged CPU instructions.
- **hardware abstraction layer virtualization** [1, 4, 9, 27]: the virtual machine monitor provides an alternate view of the hardware for the operating system. This alternate hardware view may also be invisible to the guest operating and hardware features cause certain CPU instructions to be redirected to the virtual machine monitor.
- **OS level namespace virtualization** [19, 29]: the host operating system is the virtual machine monitor. It partitions itself into multiple disjoint namespaces and gives applications in a partition the impression they are the only applications running on the operating system.

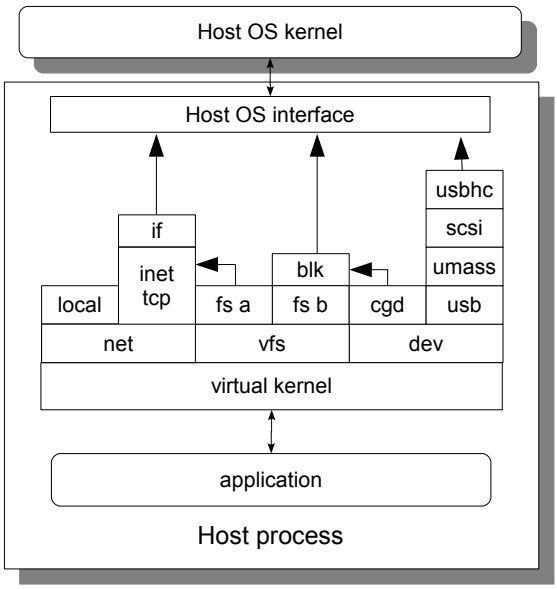

Figure 1. Runnable Userspace Meta Programs Architecture. The application and kernel services are running inside a process provided by the host operating system.

An additional noteworthy technique is API level OS virtualization. The a well-known example is the Wine project, which aims to emulate Windows APIs and allows Windows applications to work on non-Windows hosts. This, however, does not virtualize the operating system itself.

Next we outline our approach, compare it to other virtualization technologies, and analyze the implications of our choices.

2.1 Approach

As mentioned in the introduction, our approach is to take the kernel and run service components in a userspace process. We *optimize* for the case where the whole virtualized image is in a single process, but we also *support* RPC models where virtualization is spread into multiple processes on the same host or distributed amongst multiple hosts. RPC

	Full OS	Access	Host Kernel	Std. guest	Std. App	Std. HW	MI	crashproof	code
Xen	yes	guest	modified	no	yes	yes	no	yes	native
qemu	yes	guest	std.	yes	yes	yes	no	yes	translated
KVM	yes	guest	support	yes	yes	no	no	yes	native
UML	yes	guest	std.	no	yes	yes	no	yes	native
jails	no	both	support	n/a	yes	yes	yes	no	native
rump	no	both	std.	n/a	no	yes	yes	yes	native

Table 1. Comparison of popular virtualization technologies. We use specific names rather than generic names, because we believe them to be better established. The meanings of the fields are further explained in Section 2.2.

allows e.g. to use `ifconfig` and `raidctl` to configure the services in a rump kernel without having to include the same configuration functionality in every rump application.

2.2 Virtualization technology comparison

We briefly compare the key points of commonly used virtualization technologies against each other in Table 1. This is to put us into context. A more complete survey can be found elsewhere [24]. It should be noted that the technologies we include are listed using proper names. For example, another way to read "Xen" is "paravirtualization". The meanings of the fields are as follows:

- **Full OS**: does the virtualization technology virtualize every component in the OS stack.
- **Access**: where is the virtualized operating system directly accessible from for e.g. making system calls.
- **Host kernel**: does the virtualization function on a standard host kernel ("std."), does it require driver support ("support"), or does it depend on a completely different type of host kernel ("modified").
- **Std. guest**: does the technology require a specialized guest or will a normal binary distribution work.
- **Std. App**: do unvirtualized applications work directly.
- **Std. HW**: does the technology run on normal hardware or does it require special support.
- **MI**: is the implementation machine-independent, i.e. does it work with any architecture or is porting required.
- **crashproof**: is the host safe from crashing due to a problem in the virtualized portion.
- **code**: does the code execute directly on the CPU or does it require translation.

2.3 Analysis of approach

The main advantages of the rump approach are as follows:

- **minimal resource consumption**: every virtual kernel uses only what it needs
- **works on a stock host OS**: no specially modified host operating system is required, making the facility easy to take into use without prior preparation.
- **architecture-independent**: rump does not require low-level architecture-specific code and works on e.g. MIPS, ARM and PowerPC in addition to x86. Currently we have under ten lines of architecture specific code for machine architectures which support the kernel module ABI.
- **virtual kernels are first class citizens on the host**: virtual kernel access from the host OS is possible directly without having to go through a process running on the virtual kernel. This makes e.g. regression testing the kernel simpler – a coredump is an application coredump.
- **minimal setup**: setup of a virtual process kernel requires only the setup of the component it uses. It does not, for example, require creating a root file system and installing a full setup of userland utilities, if just TCP/IP virtualization is desired.

  Even if multiple different VM setups are available as pre-configured images, their management becomes hard at some point and keeping them all up-to-date can be a burden and security risk [15]. Since rump kernels are a first-class citizen of the host operating system, it is enough to keep one system up-to-date.

We drew ideas from the Exokernel [18] in that we give the application control of how to modify and optimize the kernel, Alpine [13] and Mach [16] in that we run existing kernel code in a userspace program and Denali [27] for lightweight virtualization. What sets virtual process kernels apart is that they are in fact an amalgamation of the qualities mentioned: an application-customizable and lightweight virtual kernel mechanism which uses existing operating system code and can use a stock operating system for hosting. We do not require a specially written operating system.

On the other hand, our approach is similar to kernel namespace partitioning, such as FreeBSD jails [19], Linux OpenVZ or the Featherweight Virtual Machine [29]. They all function using host operating system resources, but partition the system in a fashion so that processes belonging to a different partition cannot see or directly influence each other. However, they all operate on one copy of the kernel and its functionality. This does not provide kernel isolation, neither does it provide the ability to use multiple differing copies of the same service.

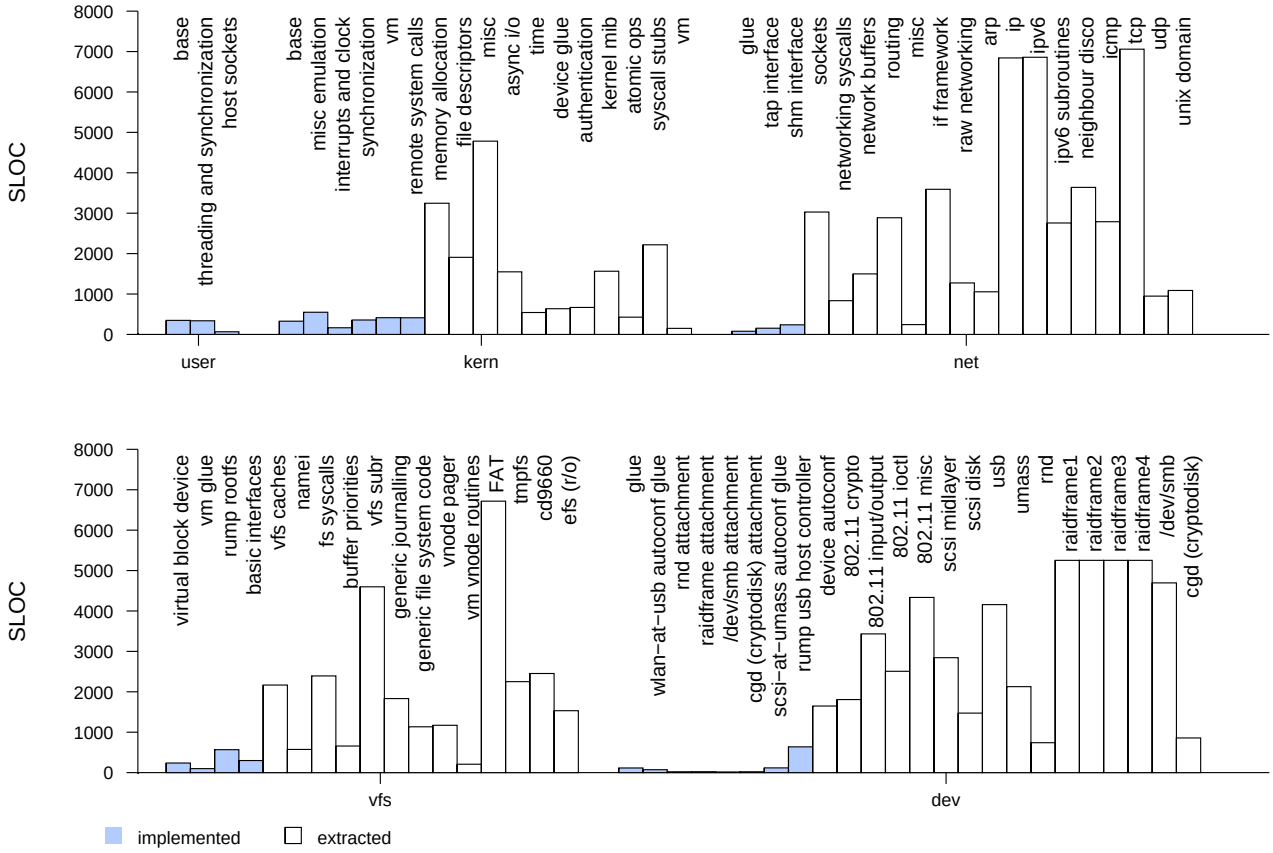

Figure 2. rump component breakdown: The source modules are classified into functional units and the lines of code for each are listed (whitespace or comments not counted). Driver examples such as TCP/IP and SCSI are also included.

3. Implementation

Rump functions by running target OS kernel code in a process on the host OS. As with all application processes, the code is running with user level privileges. The host OS controls access to physical resources such as memory, networking, devices and mass media. The rump kernel can be run with or without superuser privileges, making fine grained control of resources possible.

The rump kernel explicitly accesses the virtual machine monitor (i.e. host OS) through a special set of interfaces, known as ”rumpuser”. In Figure 1 this is depicted as ”Host OS Interface”. This component provides e.g. device access, the time of day, and synchronization primitives.

3.1 Partitioning the kernel

We partition the kernel into coarse-grained components: host OS interface, rump base, dev, vfs and net. The reason is to minimize the memory overhead of a virtual instance. We provide the memory cost of some configurations in Figure 3. Being able to cut unwanted components saves resources especially when considering thousands of units.

For example, a rump kernel supporting a USB wireless Ethernet stick has these drivers and components: dev_usbrum (the wireless driver itself), dev_net80211 (generic wireless card support), dev_usb (USB stack), dev_usbhc (rump USB host controller), dev (basic device support), net_netinet (TCP/IP), net_net (interface support), net (basic networking and sockets support), vfs (file systems support, required since the driver loads firmware from disk), crypto (generic crypto algorithm support) and rumpkern plus rumpuser.

We have two methods of providing functionality in the rump kernel: we can *extract* it out of the kernel sources, meaning we simply use the source module as such, or we can *implement* it, meaning that we do a reimplementation for userspace. We work on a source module granularity level. This approach is simple and showably works. Figure 2 provides an overview of each component, how much of extracted code there is and what we had to reimplement.

Along the way we ran into five files which we could not extract as such either because they violated component boundaries or because parts of the module accessed hardware. In these cases we split code to another module. While code moving is generally frowned upon due to it making

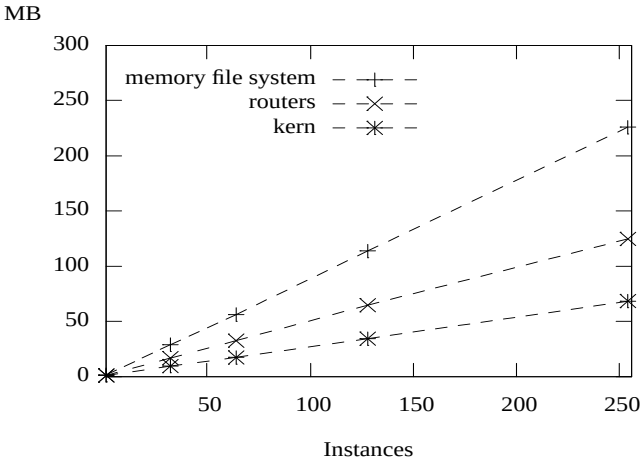

Figure 3. Memory usage: The memory usage of rump applications after startup of n instances.

changes hard to track, we feel it was an acceptable tradeoff. At least the author expected partitioning an operating system with 40 years of monolithic Unix background to be considerably harder, so the ease was a surprising result.

The rule of thumb is to extract as much as possible for the features we desire. Broadly speaking, there are three categories when extraction is not possible.

1. **code which does not exist in the kernel**: this means drivers specific for rump. Examples are the rumpuser host interface, tap interface and syscall RPC support.
2. **code dealing with concepts not supported in userspace**: the virtual memory subsystem is a premium example. We do not support page faults or secondary memory in rump. Notably though, these observations hugely simplify the rump vm.

   Nevertheless, it should be noted that parts of the vm deal with data management, e.g. manage pages associated with vnodes. These parts of the vm we want to extract.
3. **bypassed layers**: we leave e.g. thread scheduling and synchronization to the host. Any kernel code implementing bypassed layers is not applicable for rump.

3.2 Privileged instructions

Some kernel code executes CPU instructions which are available only in privileged mode. A common example is code dealing with the MMU. Executing privileged instructions while the CPU is in non-privileged mode should cause a trap and the host OS or VMM to take control. However, in some cases [25] instructions required for virtualization may pass silently through without a processor trap.

Virtualization technologies solve the problem simply by not executing privileged instructions on the CPU. For example, Xen uses hypervisor calls in user domains, UML does not use them in the usermode machine dependent code, and qemu handles such instructions in the emulator.

If we are to run unmodified kernel modules in userspace, we do not have the same freedom as any of the other technologies mentioned above. However, in practice kernel drivers do not use privileged instructions because they are found only in the architecture specific parts of the kernel. Therefore, we can solve the problem by defining that it does not exist in our model – if there are some it is a failure in modifying the OS to support rump. This subject is revisited in Section 6, where we discuss operating system requirement for rump style virtualization.

3.3 Threads and Processes

In full OS virtualization the virtual kernel takes care of thread scheduling. In the rump model we delegate this responsibility to the host.

We map kernel thread creation and the synchronization primitives to host the pthread library. This is a simple interface translation, e.g. `kthread_create()` calls `pthread_create()`. Scheduling and locking are directly done by the host and can be efficiently handled [21].

We permit applications to create processes in a rump kernel; e.g. in testing it is helpful to operate on two separate file descriptor sets. However, the whole process facility is advisory, and rump does not enforce process boundaries.

When run in a single process there is no protection for rump kernel memory. We have not found a rump use case where this is an issue. However, if one is discovered, multiple host processes and RPC into the rump kernel may be used. Another option would be to look into single address space OS protection techniques [6].

3.4 Scheduling

Modern multiprocessor operating systems use a technique called "per CPU", which means that there is a local copy of the data for each CPU. Since the current CPU is the only one with a reference to the data, a thread running on it may access the data without locking, provided that CPU preemption is disabled for the duration of the access. We must make sure that this property is preserved, or kernel code written with this assumption will not function properly.

Normally, we think of scheduling as choosing a thread to run on a CPU. However, host thread scheduling used by rump selects the running thread for us. Therefore, we schedule virtual CPUs for threads instead. In both cases the result is the same: a 1:1 mapping between the currently executing thread and a CPU.

At every rump kernel entry point we acquire a virtual CPU from a freelist. When we return to the application or make a blocking call to the host, we release the CPU and reacquire one when the call is complete. Now, even if a thread running in the rump kernel is preempted by the host thread scheduler, it will hold its locks and CPU reference. This provides the virtual rump kernel the multiprogramming illusion and ensures correct operation.

3.5 Interrupts

An interrupt is an asynchronous condition raised by the hardware. An operating system's usual response is to borrow the currently executing thread's context to examine and examine why the interrupt was caused. Many modern operating systems only acknowledge the hardware in interrupt context and defer any heavy processing to a soft interrupt which executes from thread context. Interrupt priority levels, *ipl*s, may be used to guard access to data which can be used from interrupt context. For example, in NetBSD a lock which might be taken by an interrupt must not be acquired at an ipl where an interrupt might preempt the running thread and need to take the lock the preempted thread holds.

We do not have direct hardware access in userspace and therefore do not get interrupts from the hardware. As was explained in the section on scheduling, threads running in the kernel do not get preempted from the CPU. Therefore, interrupts can always be delivered in thread context on a free virtual CPU and there is no need implement interrupt disabling like in the real kernel.

3.6 Requests to the kernel

In a normal scenario an application requests services from the kernel by doing a system call. This works by marshalling the arguments to the call in libc and trapping into the kernel. The kernel examines the system call number to decide how to retrieve the arguments and service the request.

In rump we have two distinct cases:

1. **The rump kernel is in the same address space as the application**, as is the common case. Here a system call can be relegated to making a function call. Copying data from and to the user address space is done by `memcpy()`.
2. **The rump kernel is in another process**, either on the local system or remote. This case is similar to a regular kernel. We marshal the arguments and send them over a socket along with the system call number. We then wait until the response to our request arrives.

   Copying data from and to the calling process must be handled by explicit requests back to the caller. When the server starts processing a request, it creates a task thread and sets the virtual memory context of the thread so that copy requests are directed to the RPC mechanism. The caller services the copy requests until the call returns.

For the distributed case service discovery, configuration and security are currently manually handled. Adding support for a well-known model is possible [7], but beyond the scope of this paper.

Rump also supports interfacing directly with the file system vnode interface. This is used to implement kernel file system servers [20]. Additionally, rump exports a class of interfaces which are useful only to applications with direct knowledge of rump. An example is creating a virtual process inside the rump kernel.

All of the three sets of interfaces are autogenerated from description files. This helps to verify that scheduling (Section 3.4) is properly handled at entry points. Since we do not export the application interface prototypes inside rump, it even helps in not accidentally making a call which would attempt to reschedule from inside rump. Furthermore, in the case of system calls and the vnode interface, the descriptions are the same as what the interfaces for the regular kernel are created from. This helps verify that rump interfaces behave the same as real kernel interfaces.

Rump system calls

Rump system calls have the same semantics and ABI as regular system calls, e.g. `rump_sys_open()` will upon success return a file descriptor for the path being opened; the file descriptor is a valid handle only in the rump kernel. Specifically, in case the rump kernel does not support vfs, -1 is returned and errno is set to `EOPNOTSUPP`. Initially, we required passing a reference to an integer to which errno would be set, but this API difference made adapting existing applications unnecessarily difficult. Now errno is set by rump similarly to in regular system calls.

Currently the policy for making a regular kernel system call vs. making a rump system call is entirely up to the application. In practice we have found it easy to modify applications for rump support. Most of the time it is a matter of changing a few system calls and can be done with cpp, e.g. `#define socket(a,b,c) rump_sys_socket(a,b,c)`.

3.7 On portability

Portability for a rump kernel means the ability to run a rump kernel on a non-NetBSD host OS. This is a powerful concept, as it allows running NetBSD kernel code on other operating systems and different versions of NetBSD. An example application where this is necessary is the disk encrypter described in Section 7.1.

Since we run the kernel code in userspace and our interface the host OS interfaces is very limited, is it tempting to call our solution portable. However, the claim deserves more dissection. When evaluating the portability of rump, there are two facets to consider (refer to Figure 1 if necessary):

1. **below**, i.e. rumpuser: is it possible for the host OS [version] to be different from the rump kernel OS
2. **above**, i.e. application interface: is it possible to make requests to the rump kernel from another OS [version]

To be portable from below, the rumpuser interface must be independent of system types and rely only on C99. For example, the interface must not rely on the `time_t` type being 32bit or 64bit. Neither must it depend on symbols such as `SOCK_STREAM` or `O_RDONLY` being universally constant.

Portability from above requires fulfilling the same constrains as portability from below. Additionally, it requires a common understanding of data types between the client and the rump kernel. Since the client can be on host with differ-

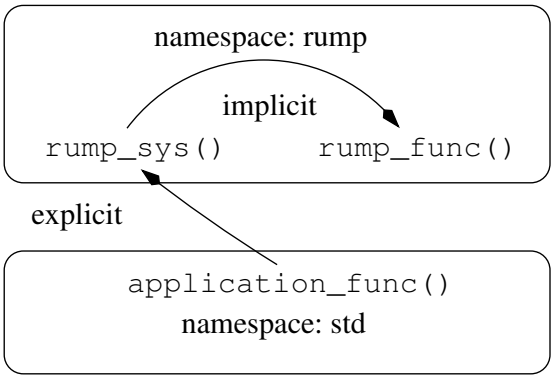

Figure 4. C namespace protection: all kernel symbols are prefixed with the string "rump" to decrease the risk of accidental collision. Calls from application namespace must explicitly include the prefix, while intra-kernel calls do not need to use the prefixed version.

ent byte order and word size and the interface larger than the rumpuser interface, the problem is much more difficult.

The issue with portability is not so much with getting the code to run as it is being able to interface with it. Since contemporary open source operating systems provide interfaces only as C headers, a readily available facility for marshalling an "on-wire" representation does not exist. It is doable, but demands a lot of manual labour and is beyond the scope of our immediate attention.

Although theoretically requiring the rigorous attention mentioned above, in practice rump is portable to some extent. Thanks to common history, for example the value of `O_RDONLY` is the same across at least Linux, BSD and Solaris systems. In practice it is possible to run rump kernels on foreign operating systems and in fact we have used NetBSD kernel file system code successfully on Linux [20]. It is also possible to host a rump kernel from a different operating system version. In fact, on all the benchmarks presented in this paper the major version of the host OS did not match the version of the rump kernel. However, there is no rule for the general case and a case-by-case study is required to determine feasibility.

Eventually, our goal is to provide both external and internal interfaces [11]. Our definition of external in this case is any system which is not ABI compatible with the rump kernel. This will preserve efficiency in the native case and allow interfacing from non-native systems.

3.8 C symbol namespaces

Normally the kernel and process C namespaces are disjoint. This means that both the kernel and an application can have and use a common symbol name, for example `printf`, without a collision occurring. When we run the kernel in a process container, we must take care to preserve this property. Otherwise the kernel version of printf would override the libc version during linktime and the calls made from inside the application would be resolved to the kernel symbol. Single address space operating systems have solved this [8], but are general for an arbitrary number of processes and require a different calling convention. In our case we only have two namespaces: application and kernel.

We solved the issue by defining all symbols starting with the prefix "rump" to be private to the rump kernel. We use the *objcopy* utility's rename functionality to prefix the kernel symbols with "rumpns_". This puts all kernel symbols into our private namespace. Now `printf` will be exported as `rumpns_printf`, and unresolved symbols will have the same prefix. Since we do not execute rename on application binaries, a single program image can contain calls to both the libc and kernel printf. Prefixes are illustrated in Figure 4.

However, renaming presents a problem. Not all symbols in an object are from the kernel. Some are a property of the toolchain. A good example is `_GLOBAL_OFFSET_TABLE_`, which is used by position independent code to store the offsets. Renaming the toolchain-generated symbol causes linking to fail, since a toolchain expects to find the offset table where it left it.

We observe that almost all of the GNU toolchain's symbols are in the namespace "__", while the NetBSD kernel exports very few symbols in that namespace. Hence, we exclude that namespace from the bulk rename. In addition, we need to maintain a quirk table. Currently, the quirk table includes only the global offset table mentioned earlier, and machine dependent quirks for PARISC and MIPS – one quirk per architecture. The solution presented here has worked well in practice and has not required adjustments since it was put in 10 months ago.

3.9 Shared Libraries vs. Kernel Modules

We support two options in rump: compile kernel code as a regular userspace shared library, or use a kernel module binary. The advantage of the latter is that it is readily available without source code or extra recompilation.

However, there is a disadvantage to using kernel modules because kernel modules are not compiled as Position Independent Code (PIC). This means that for every rump instance a separate copy of the read-only data and the text segment is loaded. This can mean significant overhead. We recommend compiling shared (PIC) libraries if source code is available.

The ability to use binary kernel modules also places restrictions on the emulated part of a rump kernel. The application binary interface (ABI) of non-extracted implementation must match the kernel ABI. This poses a problem because not all machine specific functionality is made up of function interfaces, but rather a mixture of preprocessor macros and inline functions. A good example is architecture specific memory management code. If we are to provide a userspace implementation for the headers, we must be sure to satisfy all architectures.

We support full kernel ABI compatibility currently only on i386 and x86_64. For the rest of the OS's architectures, we provide blanket headers which hide the architecture's own headers and can provide a common implementation. For i386 and x86_64 almost everything is supported by the machine independent userspace code. The MD portions reimplement only 6 symbols required by the machine dependent headers.

The final requirement for a kernel module is that it must conform to the "rumpns" namespace requirements described earlier in Section 3.8. This can be done directly on the kernel module binary using `objcopy` as described earlier and does not require the source code to be available.

Dynamic loading

Kernel modules can be dynamically loaded to add functionality to a running kernel. The application equivalent is `dlopen()` which loads and links a shared library. We can dynamically load a module compiled as a library to a rump kernel by calling dlopen on it and executing the module initialization routine.

3.10 USB device drivers

Rump offers preliminary support for running kernel USB device drivers in userspace on a NetBSD host. NetBSD exports a generic USB driver *ugen* to userspace through /dev nodes. Through the device nodes we can send and receive requests to and from the USB hardware.

At the root of each USB bus is a host controller. Due to there being multiple different types of host controllers, the NetBSD kernel USB stack defines a set of interfaces which all host controllers must implement. USB device drivers use these interfaces ("pipes") to communicate with the host controller and ultimately access the hardware.

We implement a rump USB host controller. It directs requests from drivers to the ugen device and reports results. Since ugen passes almost all requests as-is to the backing device, there is very little translation to be done. DMA is a non-issue for two reasons: first, NetBSD uses the bus_dma [26] abstraction for allocation of DMA memory, so we implement what is suitable for rump. Second, all DMA programming is done by the host controller. In our case "direct" memory access means read and write to the device node.

USB support is work-in-progress, but we can currently already mount and access a file system on a USB stick with both the USB and file system driver stacks in rump. We can also configure a wireless Ethernet USB device.

4. Evaluation

4.1 Resource control

Since a rump kernel runs purely as a process, it has access to exactly those resources the host OS gives it access to. In most cases this means file access. For example, disk based file systems can be mounted based on access to the disk file and USB drivers can access hardware depending on access to the /dev nodes for ugen devices.

Similarly, it is possible to control networking from the host OS. Even though the rump kernel contains an independent networking stack and interface configuration is done based on permission inside the rump kernel, the host OS can control outside world access with its own firewall and might for example allow only one IP address to transmit and receive data from the network.

4.2 Security and compromise analysis

One of the main applications of OS virtualization is to provide isolation and enhance security. Whether actual benefit is provided depends largely on the threat model and the partitioning of virtualized systems.

Let us assume the likely case that the kernel has faulty code in a file system driver [28] and the OS can be compromised by making it access a hostile disk image such as one on a USB device. When the kernel is compromised, in the traditional Unix model all applications running in it are compromised. It is not practical to boot an entire virtual operating system every time a file on a USB stick is to be accessed. Running a rump kernel every time is practical. In case of rump kernel compromise, direct damage is isolated to the hosting process instead of to the entire system [20].

Given that in the typical scenario there is one rump kernel per application and that the rump kernel is in fact an integral part of the application, compromise of the rump kernel from the application itself is not an issue.

A rump kernel has the same privileges as a process, so from the perspective of the host system its compromise is the same as the compromise of any other application. In case rogue applications are a concern, on most operating systems process access can be further limited by facilities such as chroot and jails [19].

Therefore, rump virtualization provides good security benefits for the tasks it is meant for. But like everything else, it depends on the security of the layers it is running on.

4.3 Micro tuning

Since our virtual machine is a part of the application, it is possible to highly tune each kernel for each application. This somewhat resembles the Exokernel motivation [14], but the main difference is that we are working with a preexisting operating system and preexisting services which we want to only microtune with little or no code modification.

As a very simple example, we mention setting the clock interrupt frequency. A service which does not need the clock, such as most file systems, can set the clock interrupt frequency very low or even completely disable it. Conversely, an application which requires high-resolution timeouts can set the clock frequency extremely high. When dealing with a high number of rump kernels this is very useful. For example, in our routing cluster, which is used to for evaluation in Section 5, activating 256 threads 1000 times a

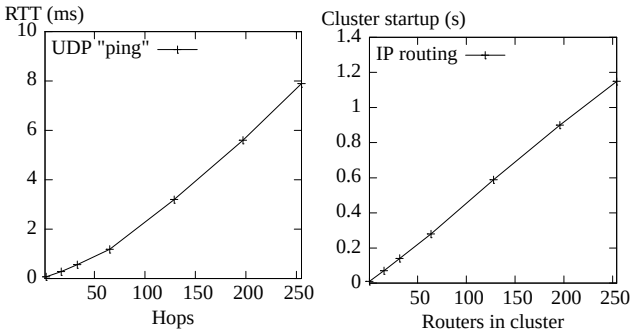

Figure 5. Packet RTT. Send an UDP packet to a peer through n nodes and wait for a response. The left hand figure is for a prerunning cluster, while the right hand figure include time to start the cluster.

second for a clock interrupt consumes close to 10% total system CPU. Setting the value as low as 5Hz is completely acceptable, since in our example only the TCP timer depends on the clock frequency. This brings idle CPU consumption down to under 0.05% of total system CPU.

A potential idea is to use profile guided optimization [5] for compiling the rump kernel. This way each application can use a specially optimized version of kernel services.

4.4 Maintainability

The maintainability of a system implemented partially in parallel with a fast-moving target such as an open source OS kernel is always suspect. Our file system paper [20] discussed maintainability. In summary, under 0.2% of kernel commits caused build breakage due to mishandling of features duplicated in rump.

5. Measurements

Almost all the measurements have been done on a 2GHz Core2Duo with 2GB of RAM. When two machines were needed, the peer was a 1.6GHz Pentium 4 over a 100MBit LAN. The rump kernel uses NetBSD 5.99.7, the Core2Duo NetBSD 5 and the Pentium 4 is running NetBSD 4.

In cases where external network access was required, the rump kernel interfaced the network through a tap interface which was bridged to the LAN interface. When simulating a purely internal network inside a single machine, a virtual interface using process shared memory as the bus was used.

5.1 Virtual network cluster

High scalability makes rump a promising option for large-scale networking testing [17] by allowing physical hosts to have multiple isolated networking stacks and routing tables.

To test the potential, we form a routed virtual network using the shared memory interface and measure the time it takes for a UDP packet to travel from one peer to another and back. The results as a function of the number of hops displayed in Figure 5. One hop corresponds to one virtual process kernel and one networking stack instance. Scaling

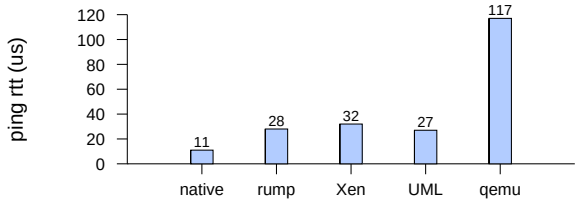

Figure 6. ping: RTT for ping with various technologies

is almost linear. We suspect the slight exponential nature is due to host OS scheduling artifacts.

The tested cluster size we tested is limited by the maximum number of hops that the IP time-to-live (TTL) field supports (255). The recommended default from RFC1340 is 64 hops, so we had to adjust the virtual kernel to default each packet to 255 hops at startup.

In addition to be highly scalable, rump kernels feature almost instantaneous startup. Using other virtualization technologies startup can take anywhere from seconds to minutes. As evidence we offer the fact that booting our standard qemu configuration used some kernel development purposes takes over 10 seconds to get to the stage to be able to respond to icmp ping and even longer to get applications running. Rump reaches the same stage in under 0.01 seconds, making it well over 1000 times faster for startup. In addition to cluster reconfiguration, fast bootstrap is important in situations where restarts are common, such as code development.

Figure 5 shows how long it takes to bootstrap our cluster we used in the previous test. To make sure we are operational, we include the time until the first packet roundtrip has been completed.

5.2 ping

We measured the latency of ping against common virtualization technologies. Since ping is handled by the kernel, there is no process scheduling involved.

The results are presented in Figure 6. Rump performs second best in the test after UserMode Linux. However, since the is no networking rump for Linux and no usermode port of NetBSD, the figures are not directly comparable – Linux might respond to ping better than NetBSD. All virtualization technologies perform significantly worse than the native case. This is because pinging a local address allows the system to route the packet through the loopback address. Since a virtualize OS does not have a local address configured on the host, the packet must be transmitted through the bridge interface and to the virtual kernel.

5.3 Web servers

A web server performance test measures two things: 1) how easy is it to adapt a real world application to use rump system calls 2) how well does rump perform in a real application. We tested thttpd and apache. Both were easy to adapt

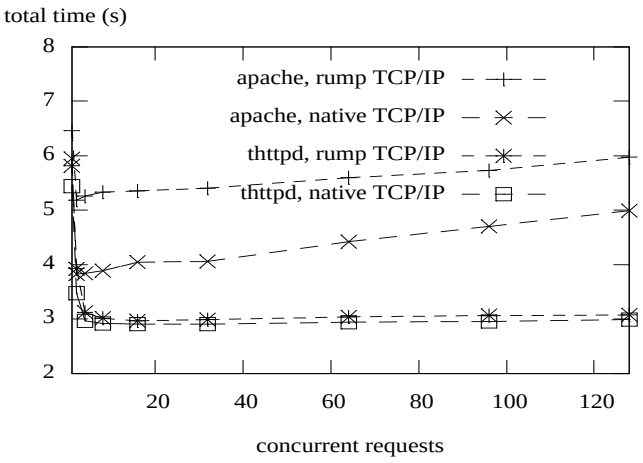

Figure 7. TCP/IP benchmark: time it to takes to perform 10000 GET requests over LAN for an 80-byte root file on a web server.

and Apache provides a separate OS portability layer ("apr") which made it easy to locate all the network access points.

We used ApacheBench to measure the total time for 10000 requests to the root document of the web server. ApacheBench is always running on the native kernel TCP/IP stack, while the web server runs both against the host stack and the virtual stack. The results are displayed in Figure 7.

Apache provides several worker models to choose from, including a fork model and a threaded model. To make a similar comparison for both thttpd and Apache, we used the fork model with a single server.

Rump TCP/IP is slower. Using thttpd with concurrency of 4 and above, the difference is about 0.1s in total time. This translates to 0.01ms (3%) difference per request. We attribute this to the fact that in addition to working the normal interface path, the virtual process kernel must deliver packets through the tap and bridge drivers. This is an issue inherent to Ethernet access from userspace. Apache does not perform nearly as well. We did not pinpoint the issue.

As an interesting observation, running ApacheBench put 10000 connections in `TIME_WAIT` on the server. This behavior is required by the active close of TCP state machine. Having 10k connections waiting forced us to wait between for the timeout between running the tests for the in-kernel networking stack. In contrast, we could kill the process hosting the rump kernel networking stack and start out with a new IP and a clean state in a fraction of a second. While native TCP/IP was faster than rump, executing the benchmarks on it took over 10 times as long in wall time.

5.4 System call speed

A common way to measure system call overhead is to execute a null system call. We wanted a call which has to copy data both from and to the user and settled on a call which looks up a value from the kernel management information base and returns it. The results are presented in Figure 8.

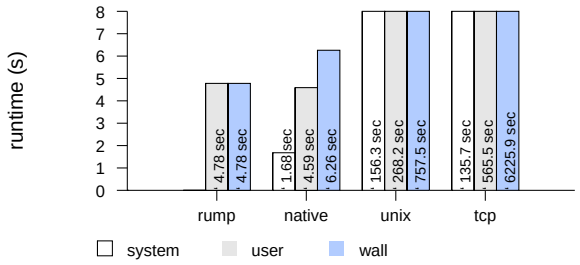

Figure 8. Syscall time: time it to takes to perform 10,000,000 simple system calls. The "unix" case depicts doing a call to a local machine kernel of a UNIX-domain socket and "tcp" does the same over host TCP/IP to a kernel running on a remote machine.

The results are expected. Making a function call is cheaper than doing a system call trap, so rumps perform better than regular applications. The "unix" and "tcp" measurements are included only as a proof-of-concept, as the remote system call mechanism is currently extremely simplistic. We have no doubt it could be dramatically improved especially in the local case by using standard microkernel techniques [3, 22]. However, doing so would mean a non-standard host OS requirement.

6. OS requirements for rump virtualization

Next we list and explain the OS requirements for implementing rump style virtualization.

- **clean source module boundaries**: we do not include all modules present in a standard kernel. If source modules have erratic interdependencies, it is difficult to extract a clean subset.
- **easy access to basic routines**: since we rename all symbols in kernel code, no symbols get linked against libc. This means that basic routines such as `memset()` and `in_cksum()` must be provided in rump code. We could provide MI implementations, but that would unnecessarily slow things down over the optimized versions. In NetBSD all the basic functionality is included in a library called *libkern*. Simply including this library in rumpkern addresses this issue.
- **autogenerated interfaces**: as mentioned in Section 3.6, the fact that NetBSD autogenerates system call tables and vnode interfaces from description files made it much simpler to wrap them with the necessary scheduler calls. Since we modified the generator scripts to produce the rump versions as well, they stay automatically in sync with the masters.
- **no static initializers**: static initializers expose implementation. For example, we map the kernel mutex type `kmutex_t` to a pthread mutex. If the kernel mutex facility

supported static initializers, a mutex would be initialized to use the kernel implementation and not our pthread-mapped implementation. Checking for lock initialization along the use-path would add unnecessary complexity and require further synchronization when two threads lock the same mutex for the first time simultaneously.

- **function interfaces**: many operating systems contain interfaces which are half implemented as inline functions or macros. For example, the interface for fetching the current running thread typically contains machine dependent code: on x86 it is fetched from a constant address behind another segment register while on UltraSPARC it is fetched from an offset in kernel VA. Sometimes, such as on x86, it is possible to satisfy the exposed implementation in userspace. Other times, such as on UltraSPARC, it is extremely complicated at best. And in any case, it requires writing machine dependent code.

  Initially when we wanted to use the kernel ABI in userspace to make kernel modules work, we discovered that on the i386 and x86_64 ports interfaces which historically were macros or inlines had been converted to function calls for kernels supporting loadable modules. This simplified our work, since we could for example relegate the management of the current thread information to pthread TLS instead of using segment registers.

We discover that apart from normal good programming practices and module separation on functionality, the kernel structural requirements for a rump virtualization are the same as those for a modularized kernel with a stable ABI.

7. Applications

In this section we present example users of rump style kernel service virtualization and compare to other solutions. The comparison summary is provided in Table 2.

7.1 Encrypted Install Images

The cryptographic disk driver (cgd) [10] provides block device encryption. The cgd driver is configured on top of a storage device and it provides an encrypted view of the underlying device. The common use case is mobile and removable storage, where a file system is created onto an encrypted view, mounted, and written to. Since cgd supports password-based PKCS#5 PBKDF2 key generation, it is possible to protect this encrypted view behind a passphrase.

Encrypted install images allow to distribute media with sensitive material such as proprietary applications, but only let parties who know the passphrase to access the sensitive part. Since the NetBSD build process is required to work as an unprivileged user on any operating system [23], relying on cgd in the build host kernel is not an option for supporting encrypted install images.

We have written a proof-of-concept application for encrypting a disk image. It configures storage into a rump kernel and a cgd device on top of it. The desired data, which can be a file system image, is then written to the cgd device node and hits the backend storage encrypted. The kernel cgd driver does not yet support in-place re-keying, but the author of cgd has expressed interest in adding support. This would simplify our application use in the future even more.

Approach	Weaknesses
OS Virtualization	• installation and maintenance • resource waste • bootstrap cost • slow or special VMM
Application	• code maintenance • not transparent
Host Kernel	• may require privileges • does not virtualize

Table 2. Summary of weaknesses using other approaches for presented applications

Application-level solutions such as pgp+tar could be used for the same effect, but they add extra steps to when the data is to be used: the encrypted package must be unencrypted and unpacked before it can be used. By using cgd the on-disk data can later be transparently accessed in NetBSD.

7.2 Self-Contained NFS Service

An NFS service is typically provided by an in-kernel file server (nfsd) and a userspace service for loading export lists and processing mount requests (mountd). Since mountd uses a SUN RPC protocol, a service binding RPC programs to network addresses is also required (rpcbind).

In environments such as labs for OS development it is desirable to let users provide their own NFS services from a centrally administrated server machine. This enables people to provide their own NFS root file systems for machines they are developing. Since nfsd runs in the kernel, it is impossible to isolate these exports from each other. Furthermore, the approach does not work for unprivileged users, since they must be able to make privileged system calls.

One way to solve this is by giving each user a virtual operating system and root inside it. This approach is overkill. It adds the administrative tasks of managing the virtual installations and upgrading them without disrupting user work. It is also a very resource-hungry approach in a situation where there are tens of simultaneous users. Another possibility is a userspace nfsd. A modern one is hard to find, though. Even if one is written, there are still issues such as port conflicts and how to be able to serve files owned by root.

We implemented a solution using rump. It combines rpcbind, mountd and the kernel nfsd, the kernel TCP/IP stack and a file system driver into a self contained application. Upon startup, the application mounts a given file system image inside the rump kernel, configures a network address and starts serving NFS requests.

8. Conclusions

We presented Runnable Userspace Meta Programs, an architecture for virtualizing services from an existing operating system kernel by using processes as containers for virtualization. The benefits are easy deployment, maintainability, speed, a lightweight nature and scalability. Supported services are networking, file systems and device drivers.

Applications of the system are in the areas of virtualization, isolation and segregation, development and debugging, testing, and optimizing kernel services for each application.

Most virtualization technologies depend on the machine architecture either in what they provide, what they can be hosted on, or both. Rump can be made to work anywhere.

We believe that this work shows the direction to how monolithic Unix kernels should be structured: it extrudes rigor and a component-oriented approach from a system which has been historically very loosely structured. We further analyzed the structural requirements of a kernel targeted for rump virtualization and found them to be akin to clean design and kernel module support. Our implementation was done and is in use on NetBSD.

Availability

The described code is available under the BSD license from the NetBSD repository in the directory `src/sys/rump`. See `http://www.NetBSD.org/` for further information.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of SOSP*, pages 164–177, 2003. ISBN 1-58113-757-5.

[2] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proc. of USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *Proc. of SOSP*, pages 102–113, 1989.

[4] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including os and network aspects. In *Proc. of IEEE International Symposium on High-Assurance Systems Engineering*, 2001.

[5] P. P. Chang, S. A. Mahlke, and W.-m. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[6] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994. ISSN 0734-2071.

[7] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proc. of the 5th MobiCom*, pages 24–35, 1999.

[8] L. Deller and G. Heiser. Linking programs in a single address space. In *Proc of. USENIX Annual Technical Conference*, pages 283–294, 1999.

[9] J. Dike. A user-mode port of the Linux kernel. In *ALS'00: Proc. of the 4th Annual Linux Showcase & Conference*, 2000.

[10] R. C. Dowdeswell and J. Ioannidis. The cryptographic disk driver. In *Proc of. USENIX Annual Technical Conference, FREENIX Track*, pages 179–186, 2003.

[11] R. Draves and S. Cutshall. Unifying the user and kernel environments. Technical Report MSR-TR-97-10, Microsoft, 1997.

[12] H. Eiraku and Y. Shinjo. Running BSD kernels as user processes by partial emulation and rewriting of machine instructions. In *Proc. of the USENIX BSD Conference*, 2003.

[13] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level infrastructure for network protocol development. In *Proc. of USENIX Symp. on Internet Technologies and Systems*, pages 171–184, 2001.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of SOSP*, pages 251–266, 1995.

[15] T. Garfinkel and M. Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Proc. of HotOS*, pages 20–20, 2005.

[16] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid. UNIX as an application program. In *Proc. of USENIX Summer*, pages 87–95, 1990.

[17] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *Proc. of USENIX Annual Technical Conference*, pages 113–128, 2008.

[18] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. of SOSP*, pages 52–65, 1997.

[19] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proc. of SANE Conference*, 2000.

[20] A. Kantee. Rump file systems: Kernel code reborn. In *Proc of. USENIX Annual Technical Conference*, pages 201–214, 2009.

[21] S. T. King, G. W. Dunlap, and P. M. Chen. Operating system support for virtual machines. In *Proc. of USENIX Annual Technical Conference*, 2003.

[22] J. Liedtke. Improving IPC by kernel design. In *Proc. of SOSP*, 1993.

[23] L. Mewburn and M. Green. build.sh: Cross-building NetBSD. In *Proc. of USENIX BSD Conference*, pages 47–56, 2003.

[24] S. Nanda and T. Chiueh. A survey of virtualization technologies. Technical Report TR-197, Stony Brook, 2005.

[25] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proc. of USENIX Security Symposium*, 2000.

[26] J. Thorpe. A machine-independent DMA framework for NetBSD. In *Proc. of USENIX Annual Technical Conference (FREENIX track)*, pages 1–12, 1998.

[27] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *Proc. of OSDI*, pages 195–209, 2002.

[28] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *SP '06: Proc. of 2006 IEEE Symp. on Security and Privacy*, pages 243–257. IEEE Computer Society, 2006.

[29] Y. Yu, F. Guo, S. Nanda, L.-c. Lam, and T.-c. Chiueh. A feather-weight virtual machine for windows applications. In *Proc. of Virtual Execution Environments*, pages 24–34, 2006.