Towards a Flexible, Lightweight Virtualization Alternative

David C. van Moolenbroek
VU University Amsterdam
david@minix3.org

Raja Appuswamy
VU University Amsterdam
raja@minix3.org

Andrew S. Tanenbaum
VU University Amsterdam
ast@cs.vu.nl

ABSTRACT

In recent times, two virtualization approaches have become dominant: hardware-level and operating system-level virtualization. They differ by where they draw the *virtualization boundary* between the virtualizing and the virtualized part of the system, resulting in vastly different properties. We argue that these two approaches are extremes in a continuum, and that boundaries in between the extremes may combine several good properties of both. We propose abstractions to make up one such new virtualization boundary, which combines hardware-level flexibility with OS-level resource sharing. We implement and evaluate a first prototype.

Categories and Subject Descriptors

D.4.0 [**Operating Systems**]: General

General Terms

Design, Performance, Reliability, Security

1. INTRODUCTION

The concept of virtualization in computer systems has been around for a long time, but it has gained widespread adoption only in the last fifteen years. It is used to save on hardware and energy costs by consolidating multiple workloads onto a single system without compromising on isolation, as well as to create host-independent environments for users and applications.

In these recent times, two virtualization approaches have established themselves as dominant: hardware-level and operating system-level (OS-level) virtualization. The two are fundamentally different in where they draw the *virtualization boundary*: the abstraction level at which the virtualized part of the system is separated from the virtualizing infrastructure. The boundary for hardware-level virtualization is low in the system stack, at the machine hardware interface level. For OS-level virtualization it is relatively high, at the operating system application interface level. This boundary determines important properties of the virtualization system: the former is generally thought of as more flexible and better isolated; the latter as faster and more lightweight.

In this paper, we take a top-down look at the virtualization boundary. We argue that the two existing approaches are extremes in a continuum with a relatively unexplored yet promising middle ground. This middle ground offers the potential to combine several of the good properties from both sides. We propose a set of "mid-level" abstractions to form a new virtualization boundary, where the virtualizing infrastructure provides object-based storage, page caching and mapping, memory management, and scheduling, whereas any higher-level abstractions are implemented within each virtual environment. We argue that this approach offers a viable alternative, providing flexible, lightweight virtual environments for users and applications. We implement the core of our ideas in a microkernel-based prototype.

The rest of the paper is laid out as follows. In Sec. 2 we describe virtualization boundaries as a continuum. In Sec. 3 we propose and discuss a new alternative. We describe our prototype in Sec. 4 and its evaluation in Sec. 5. Sec. 6 lists related work. We conclude in Sec. 7.

2. VIRTUALIZATION AS A CONTINUUM

Reduced to its most basic form, any virtualization system consists of two parts. The **domains** are the environments being virtualized, with (at least) user applications in them. The **host** comprises all system components facilitating the virtualization of such domains. We refer to the abstraction level defining the separation between these parts as the **virtualization boundary**.

In this paper, we consider only virtualization in which unchanged applications can run using machine-native instructions (as opposed to e.g. Java [19] and NaCl [60]), and in which domains are considered untrusted. In this space, two approaches have become dominant: hardware-level virtualization (Sec. 2.1) and OS-level virtualization (Sec. 2.2). We argue that these are extremes in a continuum of virtualization boundaries, in which new alternatives have the potential to combine good properties of both (Sec. 2.3).

2.1 Hardware-level virtualization

Hardware-level virtualization [4,6,17,58] places the virtualization boundary as low in the system stack as practically feasible. A host layer (*virtual machine monitor* or *hypervisor*) provides its domains (*virtual machines*) with abstrac-

ACM SYSTOR '14, June 10-12 2014, Haifa, Israel
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2920-0/14/06 ...$15.00.
DOI 10.1145/2611354.2611369

tions that are either equal to a real machine or very close to it: privileged CPU operations, memory page tables, virtual storage and network devices, etc. The low boundary allows a full software stack (OS and applications) to run inside a domain with minimal or no changes. The result is strong isolation, and full freedom for the OS in each domain to implement arbitrary high-level abstractions.

However, the OS adds to the domain's footprint, while typically exploiting only a part of its flexibility. Several fundamental abstractions are common across OSes: processes, storage caches, memory regions, etc. Reimplementing these in isolation leads to duplication and missed opportunities for global optimization. Only the host side can solve these issues, but the low boundary creates a *semantic gap* [8]: the host lacks necessary knowledge about the higher-level abstractions within the domains. Many ad-hoc techniques have been proposed to work around this gap [14,26,35,39,40,55].

2.2 Operating system-level virtualization

In contrast, with OS-level virtualization [28,42,44,48,61], the operating system itself has been modified to be the virtualization host. The domains (*containers*) consist of application processes only–all system functionality is in the OS. Each domain gets a virtualized view of the OS resources: the file system hierarchy, process identifiers, network addresses, etc. Since the OS doubles as the host, there is no redundancy between domains and resources can be optimized globally. Thus, OS-level virtualization is relatively lightweight.

However, merging the host role into the OS has downsides as well. First, it eliminates all the flexibility found in hardware-level virtualization: the domains have to make do with the abstractions offered by the OS. Second, the merge removes an isolation boundary; failures and security problems in the OS may now affect the entire system rather than a single domain.

2.3 The case for new alternatives

The placement of the virtualization boundary in the software stack clearly has important consequences. However, we argue that the two described approaches are extremes in a continuum. With the boundary as low in the software stack as possible, hardware-level virtualization represents one end of the spectrum. OS-level virtualization represents the other end, with the boundary as high as possible without affecting applications. That leaves a wide range of choices in between these extremes.

In a middle-ground approach, the host layer provides abstractions to its domains that are higher-level than those of hardware, and yet lower-level than those of OSes. Each domain then contains a *system layer* which uses those "mid-level" abstractions to construct the desired interface for its applications, as illustrated in Fig. 1. This way, we have the potential to reduce the footprint of the domains while retaining much of their flexibility and isolation. The only point that we must compromise on, is the ability to run existing operating systems.

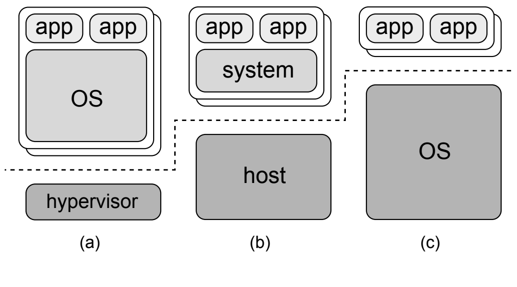

Figure 1: A schematic diagram of (a) hardware-level virtualization, (c) OS-level virtualization, and (b) our new alternative. The dashed line shows the virtualization boundary.

3. A NEW VIRTUALIZATION DESIGN

In this section, we present the design of a new point in the virtualization continuum. We first state our design goals (Sec. 3.1). We then present the abstractions making up the new virtualization boundary (Sec. 3.2). Finally, we discuss the properties of our design (Sec. 3.3).

3.1 Design goals

Our goal is to establish a new set of abstractions implemented in the host system and exposed to the domains. Each domain's system layer may use those abstractions to construct an interface for its applications. In principle, the domain should be able to achieve binary compatibility with existing applications. With that constraint as a given, we set the following subgoals.

First, the abstraction level should be high enough to bridge the semantic gap. In particular, the new abstractions should allow for **lightweight domains**, mainly by reducing resource duplication. In many cases, the domains running on a single host will bear great similarity, because they implement the same application interface, and thus can share programs, libraries, and data, both in memory and on disk. The abstractions should make sharing of such resources a natural result rather than an afterthought, but with copy-on-write (CoW) semantics to retain full isolation. Bridging the semantic gap should also allow for other **global optimizations** generally found in OS-level virtualization only.

Second, the new abstractions should be sufficiently low-level to give the system layer in each domain substantial **flexibility** in implementing (or omitting) high-level abstractions for its applications. The host system should expose only abstractions that are well established as fundamental building blocks for operating systems and thus practically used in all domains.

Third, it should be possible to provide the new abstractions through a **minimal interface**. The virtualization boundary doubles as a boundary for fault and security isolation. A simple, narrow interface reduces complexity and security risks in the host system.

3.2 Abstractions

Based on these goals, we propose the following abstractions, starting with storage and going from there.

3.2.1 Object-level storage

For storage, hardware-level virtualization typically exposes a block-level *virtual disk* abstraction, while OS-level virtualization typically exposes a full-blown *file system* abstraction. We argue that neither is optimal.

Block-level storage again suffers from a semantic gap. Since algorithms at the block level lack filesystem-level information, block-level storage is plagued by fundamental reliability issues [33] and missed opportunities for optimization

[10, 27], also in the context of virtualization [51]. In addition, virtual disks are intrinsically heavyweight: they are statically sized and the smallest unit of content sharing between domains is the whole virtual disk.

However, file systems represent the other end of the spectrum. They impose a specific naming structure and constraints, thus adding host complexity while taking away flexibility from the domain, in terms of semantics (e.g., POSIX vs Win32 file deletion), configuration (e.g., access time updates or not), and the ability to optimize metadata management for application needs [52].

Instead, we propose an object storage model [12,16,54], in which the host exposes an abstraction of *objects*: variable-size byte containers, each with a unique identifier and a set of associated attributes. An object can be created, deleted, read from, written to, truncated, and have its attributes be retrieved and manipulated. This small set of operations makes up the core of the storage interface implemented by an **object store** in the host.

The object model imposes no structure *between* objects. It is left to the system layer in each domain to tie together its objects into a namespace. The basic approach is to use one object per file in the file system. Directories may be implemented as one object each, or with centralized objects [52], for example. With no hierarchy or metadata management defined at the object level, each domain may implement the file system abstractions appropriate for its user applications.

The details of storage management are left to the centralized object store. This store maintains and persists the global set of objects, on local or possibly networked storage. It gives each domain its own virtualized view of the set of objects, by keeping a per-domain *mapping* of local object identifiers to global objects.

The mapping facilitates storage space sharing similar to block-level CoW storage. The object store can map several per-domain identifiers to a single underlying object. Upon creation, a domain's initial mapping typically points to a set of preexisting objects. Such objects remain shared until a domain changes them, at which point it gets a private copy, thus retaining full isolation. As a result, the domains' object spaces will only ever diverge; to overcome this, the store can employ lightweight object-level deduplication to merge objects that become identical later. It can also implement finer-grained sharing and deduplication, for which it can use object-level hints.

Thus, overall, the object-based storage abstraction enables both per-domain metadata management flexibility and fine-grained cross-domain storage sharing. At the same time, the storage implementation details are fully confined to the host side, where the object-level information can be used to improve global decisions.

3.2.2 Object page caching and mapping

Next, we propose to extend the consolidation of shared storage objects to memory, in the form of a centralized **page cache**. Positioning the page cache in the host's object store yields several advantages. First, cache pages can be associated with their underlying global object, thus avoiding in-memory duplication of cached data between domains altogether. Second, a centralized page cache can employ global optimization strategies such as domain working set size estimation [26,30,37,38] and exclusive caching on a second-level SSD cache.

With the page cache on the host side, the final step is to allow the cached pages to be CoW-mapped into domains, so as to let domains implement support for memory-mapped files. As a result, if multiple domains map pages from the same underlying global objects, these domains all end up with a copy-on-write mapping to the same physical page. This allows memory sharing of application and library code in particular.

Overall, the caching and sharing eliminates a substantial fraction of interdomain memory redundancy [7, 31] without the expense of explicit deduplication.

3.2.3 Address spaces, threads, and IPC

As supporting infrastructure, we propose that the host expose a number of microkernel-inspired abstractions: address spaces, threads of execution, and interprocess communication (IPC) [22, 50].

Thus, the host side becomes fully responsible for maintaining virtual memory and scheduling. It exposes an interface that allows for memory mapping, unmapping, granting, sharing, and copying, as well as creation and manipulation of threads. Centralized memory management not only simplifies the interface to our proposed memory mapping abstraction, but also facilitates centralized page fault and swap handling without the problem of double paging [18]. Centralized scheduling allows for global optimizations as well [29,34,49].

From the perspective of the host, each domain now consists of one or more processes making up its system layer, and a set of processes making up its application layer. In order to let these entities communicate, the host exposes IPC primitives, with access restricted as appropriate. The IPC primitives may also be used to implement additional required functionality for the domain system layer, such as timers and exceptions. We refrain from defining the exact primitives; such choices should be driven by low-level performance considerations and may be platform dependent.

3.2.4 Other abstractions

Physical resources typically require a driver and appropriate multiplexing functionality in the host. The main remaining resource is networking. Because of the state complexity of networking protocols, we believe the TCP/IP stack should be inside the domains. The abstractions exposed by the host can therefore simply be in terms of "send packet" and "receive packet," implemented by a host-side network packet multiplexer.

3.3 Properties

We now discuss several properties of our proposed virtualization approach and its implementations.

3.3.1 Flexibility and security

The system layer of each domain is free to implement any abstractions not exposed by the host. For a typical POSIX domain, this would include abstractions such as: process identifiers and hierarchies; signals; file descriptors; the file system hierarchy; pseudoterminals; sockets; network and pseudo file systems. A domain may implement the minimal subset needed by its applications to minimize the domain's footprint and its system layer's attack surface, or a set of experimental abstractions to cater to an esoteric application's needs, or a specific set of abstractions for compatibility with a legacy application, etc.

Compared to OS-level virtualization, our approach shares two advantages with hardware-level virtualization. First, a privileged user gets full administrative control over her domain, including the ability to load arbitrary extensions into the domain's system layer. Second, the host abstractions are available only to the domain's system layer, resulting in two-level security isolation: in order for an unprivileged attacker to escape from her domain, she would first have to compromise the domain's system layer, and then compromise the host system from there.

3.3.2 Performance versus subsystem isolation

Our design imposes no *internal* structure on the system layers on either side of the virtualization boundary. The host side may be implemented as a single kernel, or as a microkernel with several isolated user-mode subsystems. Independently, the system layer of each domain may be implemented as a single user-mode process, or as multiple isolated subsystems. On both sides this is a tradeoff between performance and fault isolation.

We leave open whether a monolithic implementation on both sides can achieve low-level performance on par with other virtualization architectures. For example, since the page cache is in the host system, an application has to go through its domain's system layer to the page cache in order to access cached data. In a straightforward implementation, this would several extra context switches compared to the other architectures. Future research will have to reveal to which extent any resulting overheads can be alleviated, for example with a small domain-local cache, memory mapping based approaches, or asynchronous operations [24, 41, 47].

For the host system, the resulting size of the trusted computing base (TCB) will be somewhere between that of hardware-level and OS-level virtualization. A microkernel implementation would allow for a minimal TCB for security-sensitive domains by isolating them from unneeded host subsystems, similar to Härtig et al [21].

3.3.3 Resource accounting and isolation

A proper virtualization system must give each domain a share of the system resources and prevent interference between domains [4, 20, 36, 48]. This problem is smaller when the host side provides fewer services: any resources used within a domain can easily be accounted to that domain, but the host must account its own resource usage to the appropriate domain explicitly. Our approach takes a middle ground: explicit accounting is needed for the object store and the memory manager, but not for higher-level resources. We expect to be able to leverage existing OS-level virtualization algorithms.

3.3.4 Checkpointing and migration

For common virtualization functionality such as checkpoint/restart and live migration [9, 42], our solution benefits from the middle ground. Compared to OS-level virtualization, our host system provides fewer abstractions, and therefore less state to extract and restore explicitly. Compared to hardware-level virtualization, our central memory and storage management simplify the solution. The domains do not manage their own free memory, and any domain-specific pages in the page cache can be evicted at any time. Thus, the footprint of a domain can be minimized by the host at any time, obviating the need for ballooning [55].

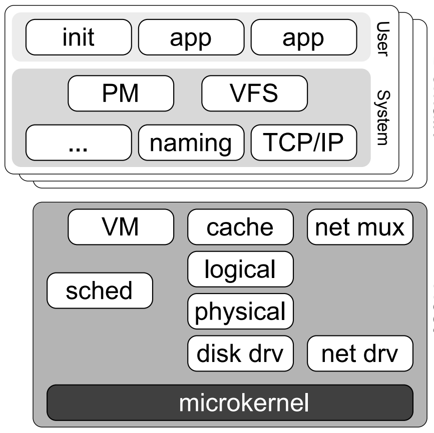

Figure 2: The components of our prototype.

4. OUR PROTOTYPE

We have built an initial prototype on the MINIX 3 microkernel operating system [50]. MINIX 3's microkernel implements a small set of privileged operations. This set includes low-level IPC facilities, which identify processes by global *endpoint* numbers. On top of the microkernel, POSIX abstractions are implemented in isolated user-mode processes called *services*. The most important services are: VM, the virtual memory service, which manages physical memory and creates page tables; SCHED, a scheduling policy service; PM, the process manager, which manages process IDs and signals; and VFS, the virtual file system service, which manages open file descriptors, working directories, and mounted file systems. PM and VFS provide the main POSIX abstractions to application processes, whereas VM and SCHED are not exposed to applications directly. Other services provide file systems, a TCP/IP stack, pseudoterminals, SysV IPC, etc. Hardware interaction is made possible with device driver services.

In prior work, we have developed a new object-based storage stack called Loris [2], which replaces the traditional file system and RAID layers positioned between VFS and disk drivers with four new, separate layers. Below VFS, the *naming* layer implements VFS requests, file naming, directories, and attributes. It stores files and directories as objects, using the object store implemented in the layers below: the *cache* layer, an object-based page cache; the *logical* layer, which adds RAID-like per-object redundancy; and, the *physical* layer, which manages the layout of underlying devices on a per-device basis and converts object operations to block operations. The object store uses the model and operations from Sec. 3.2.1.

We modified MINIX 3 and Loris to implement our virtualization concept. A virtualization boundary is drawn between the system services such that the host system consists of the microkernel, the VM and SCHED services, the lower three Loris layers, and all hardware driver services. Each domain has a system layer consisting of a private instance of PM, VFS, the Loris naming layer, and any of the other POSIX services as needed by its applications. The user application layer of the domain consists of a copy of the `init` user process and any actual application processes. The result is shown in Fig. 2.

Since endpoints were both global and hardcoded (e.g., PM has endpoint 0), we modified the kernel to virtualize endpoints using a mapping between global and domain-local endpoints. Thus, each PM instance has a domain-local endpoint of 0, but all PM instances have different global endpoints. The mapping also determines which host services are visible to each domain. We note that a microkernel with relativity of reference for IPC (e.g., object capabilities) would not require such a change.

We modified the Loris cache layer to support object-granular copy-on-write. To this end, we added per-domain mappings from domain-local to global object identifiers, as explained in Sec. 3.2.1, and global-object reference count tracking. An out-of-band interface is used to create and delete mappings for domains. We did not change the VM service in a similar way, as it already had support for copy-on-write memory.

We did change VM, SCHED, and the Loris cache to be domain-aware. These host services use a shared kernel page to look up any caller's domain by endpoint. In addition, we wrote a virtual LAN network multiplexer and a multiplexing console driver for the host system. We have not implemented support for resource isolation, checkpointing, migration, or storage deduplication at this time.

For implementation simplicity, there is an administrative domain which is able to set up other domains. Creating a domain consists of loading an initial object mapping into the Loris cache. Starting a domain consists of allocating a domain in the kernel, loading a bootstrapping subset of system services and `init`, mapping these processes into the new domain, and starting them.

5. EVALUATION

For now, we believe that publishing microbenchmarks is not useful: neither MINIX 3 nor our new changes have been optimized for low-level performance. Instead, we provide statistics on memory usage and sharing, startup times, and storage macrobenchmark results. We used an Intel Core 2 Duo E8600 PC with 4 GB of RAM and a 500 GB 7200RPM Western Digital Caviar Blue SATA disk.

We tested our system with 250 full POSIX domains at once. Each domain consists of 13 service processes that make up the domain's system layer, plus `init`, a login daemon, and an OpenSSH daemon. The domains can all access (only) the following host services: VM, SCHED, the Loris cache layer, the virtual LAN driver, and the console multiplexing driver.

Each of these domains uses 3148 KB of *data* memory, which includes process heaps and stacks in the domain as well as host-side memory such as kernel process structures and page tables. In addition, the domains use 2944 KB of *text* memory, which is shared between all domains. Even with only around 3.7 GB of the machine's total memory being usable due to device mappings in 32-bit mode, the described setup leaves around 3.0 GB of memory available for additional applications and caching.

Creating and loading a CoW object mapping for a new domain, based on a subtree of the administrative domain's file tree (emulating `chroot`), takes 32 ms. Starting a domain to its login prompt, including OpenSSH, takes 53 ms. Shutting down a domain takes 4 ms.

We used storage macrobenchmarks to evaluate the performance impact of our main changes. Table 1 shows absolute performance numbers for unmodified MINIX 3, and relative numbers for the kernel's new endpoint virtualization (EV) and EV combined with CoW support in the Loris cache (EV,C). For the OpenSSH and AppLevel (MINIX 3) build benchmarks, lower is better. For FileBench's fileserver and webserver, higher is better. The exact configurations are described elsewhere [54].

Table 1: Macrobenchmark results

Benchmark	Unit	Baseline	EV	EV,C
OpenSSH	seconds	393	1.01	1.02
AppLevel	seconds	621	1.01	1.01
FileServer	ops/sec	1178	0.98	0.98
WebServer	ops/sec	13717	0.97	0.96

The benchmarks show some overhead, caused mainly by new CPU cache and TLB misses. Optimizations should reduce these; all algorithms are constant-time.

6. RELATED WORK

Roscoe et al [46] argue in favor of revisiting the hardware-level virtualization boundary. However, so far, proposals to expose higher-level abstractions [11,43,45,59] generally keep the low-level boundary in order to support existing OSes. Decomposing monolithic OSes [15, 41] may eventually help port them to our architecture.

One exception is Zoochory [25], a proposal for a middle-ground split of the storage stack in virtualized environments. However, the proposal focuses on virtualization-unrelated advantages of rearranging the storage stack–ground we have covered in previous work as well [2, 3, 52]. They propose that the host system's storage interface be based on content-addressable storage (CAS); our object interface would allow but not enforce that the host side of the storage stack implement CAS. However, our redesign goes beyond the storage interface alone, and thus requires more invasive changes.

Our work shows similarity to several microkernel-based projects. L^4Linux uses task and memory region abstractions to separate virtualized processes from a virtualized Linux kernel [22]. Several L4-based projects combine microkernel features with virtualization without further changing the virtualization boundary [21, 23, 56]. The Hurd OS has "subhurds" [1]: virtual environments with long-term plans similar to ours [5]. Fluke [13] shares several goals with our approach, but its support for recursion requires high-level abstractions to be defined at the lowest level. New microkernel-based abstractions are also used for multicore scalability [32, 57].

We have previously presented an early sketch of our idea [53], focusing instead on its potential to improve reliability.

7. CONCLUSION

In this paper, we have established a new point in the spectrum of virtualization boundaries. Our alternative appears to have sufficiently interesting properties to warrant further exploration. However, with respect to virtualization architectures, we believe that the final word has not been said, and we encourage research into other alternatives in this design space.

8. ACKNOWLEDGMENTS

This research was supported in part by European Research Council Advanced Grant 227874.

9. REFERENCES

[1] GNU Hurd – subhurds. http://www.gnu.org/software/hurd/hurd/subhurd.html.

[2] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Loris - A Dependable, Modular File-Based Storage Stack. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 165–174. IEEE, 2010.

[3] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Flexible, modular file volume virtualization in Loris. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–14. IEEE, 2011.

[4] 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 *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, 2003.

[5] O. Buddenhagen. Advanced Lightweight Virtualization. http://tri-ceps.blogspot.com/2007/10/advanced-lightweight-virtualization.html, 2007.

[6] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, Nov. 1997.

[7] C.-R. Chang, J.-J. Wu, and P. Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 244–249. IEEE, 2011.

[8] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS '01, pages 133–, 2001.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, 2005.

[10] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *The Proc. of the USENIX Ann. Tech. Conf. (USENIX '02)*, pages 177–190, June 2002.

[11] H. Eiraku, Y. Shinjo, C. Pu, Y. Koh, and K. Kato. Fast networking with socket-outsourcing in hosted virtual machine environments. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 310–317. ACM, 2009.

[12] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: The future building block for storage systems. In *Proceedings of the 2005 IEEE International Symposium on Mass Storage Systems and Technology*, LGDI '05, pages 119–123, 2005.

[13] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the second USENIX symposium on Operating systems design and implementation*, OSDI'96, pages 137–151, 1996.

[14] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

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

[16] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 92–103, 1998.

[17] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[18] R. P. Goldberg and R. Hassinger. The double paging anomaly. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*, pages 195–199. ACM, 1974.

[19] J. Gosling. *The Java language specification.* Addison-Wesley Professional, 2000.

[20] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, 2006.

[21] H. Härtig. Security Architectures Revisited. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 16–23, 2002.

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

[23] G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, pages 19–24, 2010.

[24] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum. Keep net working-on a dependable and fast networking stack. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[25] W. Jannen, C.-C. Tsai, and D. E. Porter. Virtualize storage, not disks. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 3–3, 2013.

[26] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 14–24, 2006.

[27] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A File System for Virtualized Flash Storage. In *FAST'10: Proc. of the Eighth USENIX Conf. on File and Storage Technologies.* USENIX Association, 2010.

[28] P.-H. Kamp and R. N. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[29] D. Kim, H. Kim, M. Jeon, E. Seo, and J. Lee. Guest-aware priority-based virtual machine scheduling for highly consolidated server. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 285–294, 2008.

[30] H. Kim, H. Jo, and J. Lee. XHive: Efficient Cooperative Caching for Virtual Machines. *IEEE Trans. Comput.*, 60(1):106–119, Jan. 2011.

[31] J. F. Kloster, J. Kristensen, and A. Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. Technical report, Aalborg University, 2007.

[32] K. Klues, B. Rhoden, Y. Zhu, A. Waterman, and E. Brewer. Processes and resource management in a scalable many-core OS. HotPar '10, 2010.

[33] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. Parity lost and parity regained. In *Proc. of the Sixth USENIX conf. on File and storage technologies*, FAST'08, pages 1–15, 2008.

[34] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 93–102, 2012.

[35] J. R. Lange and P. Dinda. SymCall: Symbiotic Virtualization Through VMM-to-guest Upcalls. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 193–204, 2011.

[36] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *Selected Areas in Communications, IEEE Journal on*, 14(7):1280–1297, 1996.

[37] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *Usenix Annual Technical Conference*, pages 29–43, 2007.

[38] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Paravirtualized paging. In *Proceedings of the First Conference on I/O Virtualization*, WIOV'08, pages 6–6, 2008.

[39] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa. XLH: More Effective Memory Deduplication Scanners Through Cross-layer Hints. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 279–290, 2013.

[40] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: enlightened page sharing. In *Proceedings of the 2009 conference on USENIX ATC.* USENIX Association, 2009.

[41] R. Nikolaev and G. Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 116–132, 2013.

[42] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 361–376, 2002.

[43] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 26–26, 2006.

[44] D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA*, pages 241–254, 2004.

[45] H. Raj and K. Schwan. O2S2: Enhanced Object-based Virtualized Storage. *SIGOPS Oper. Syst. Rev.*, 42(6):24–29, Oct. 2008.

[46] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and virtue. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, page 4. USENIX Association, 2007.

[47] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–8. USENIX Association, 2010.

[48] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proc. of the Second ACM SIGOPS/EuroSys European Conf. on Computer Systems*, EuroSys'07, pages 275–287, 2007.

[49] H. Tadokoro, K. Kourai, and S. Chiba. A secure system-wide process scheduler across virtual machines. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 27–36. IEEE, 2010.

[50] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation (Third Edition).* Prentice Hall, 2006.

[51] V. Tarasov, D. Jain, D. Hildebrand, R. Tewari, G. Kuenning, and E. Zadok. Improving I/O Performance Using Virtual Disk Introspection. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

[52] R. van Heuven van Staereling, R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Efficient, Modular Metadata Management with Loris. In *Proc. 6th IEEE International Conference on Networking, Architecture and Storage*, pages 278–287. IEEE, 2011.

[53] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum. Integrated End-to-End Dependability in the Loris Storage Stack. In *HotDep*, 2011.

[54] D. C. van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum. Transaction-based Process Crash Recovery of File System Namespace Modules. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing*, 2013.

[55] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[56] C. Weinhold and H. Härtig. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 81–93, 2008.

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

[58] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, 2002.

[59] M. Williamson. XenFS, 2009.

[60] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[61] Y. Yu, F. Guo, S. Nanda, L.-c. Lam, and T.-c. Chiueh. A Feather-weight Virtual Machine for Windows Applications. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 24–34, 2006.