A Case for Scaling Applications to Many-core with OS Clustering

Xiang Song Haibo Chen Rong Chen Yuanxuan Wang Binyu Zang

Parallel Processing Institute, Fudan University

{xiangsong, hbchen, chenrong, yxwang1987, byzang}@fudan.edu.cn

Abstract

This paper proposes an approach to scaling UNIX-like operating systems for many cores in a backward-compatible way, which still enjoys common wisdom in new operating system designs. The proposed system, called Cerberus, mitigates contention on many shared data structures within OS kernels by clustering multiple commodity operating systems atop a VMM, and providing applications with the traditional shared memory interface. Cerberus extends a traditional VMM with efficient support for resource sharing and communication among the clustered operating systems. It also routes system calls of an application among operating systems, to provide applications with the illusion of running on a single operating system.

We have implemented a prototype system based on Xen/Linux, which runs on an Intel machine with 16 cores and an AMD machine with 48 cores. Experiments with an unmodified MapReduce application, dbench, Apache Web Server and Memcached show that, given the nontrivial performance overhead incurred by the virtualization layer, Cerberus achieves up to 1.74X and 4.95X performance speedup compared to native Linux. It also scales better than a single Linux configuration. Profiling results further show that Cerberus wins due to mitigated contention and more efficient use of resources.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Experimentation, Performance

Keywords Multicore, Scalability, OS Clustering

1. Introduction

Scaling UNIX-like operating systems on shared memory multicore or multiprocessor machines has been a goal of system researchers for a long time. Currently, there is a debate on the approach to scaling operating systems: designing new operating systems from scratch (e.g., Corey [Boyd-Wickizer 2008], Barrelfish [Baumann 2009] and fos [Wentzlaff 2008]); or continuing the traditional path of refining commodity kernels by iteratively eliminating bottlenecks using both traditional parallel programming skills or new data structures (e.g., RCU [McKenney 2002], Sloppy Counter [Boyd-Wickizer 2010]). However, with continual growth of the number of cores in a single machine and the still speculative structure of future many-core machines, there is currently no conclusion on the best long-term direction.

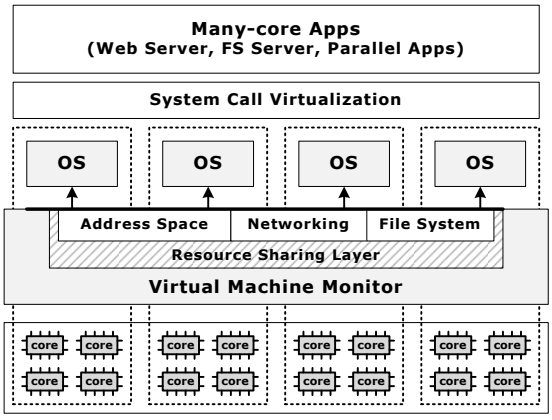

Figure 1. *Architecture overview of OS Clustering.*

In this paper, we seek to add a point to the debate, by evaluating a middle ground between these two trends, motivated by the observation that commodity operating systems can scale well with a small number of CPU cores, and one virtual machine monitor (VMM) can effectively consolidate multiple operating systems. The proposed approach, called OS clustering (shown in Figure 1), is an operating system structuring strategy that attempts to provide a near- or middle-term solution to mitigate the scalability problem of commodity operating systems, yet without non-trivial testing efforts and possible backward compatibility issues in new operating system designs. The basic idea is clustering multiple commodity operating systems atop a VMM to serve one application, while providing the familiar *POSIX programming*

EuroSys'11, April 10–13, 2011, Salzburg, Austria.

interface to shared-memory applications. The resulting system, called Cerberus, supports existing many-core applications with little or no porting effort.

The goal of Cerberus is to make a bridge between two different directions (i.e., designing new OSes and refining commodity OSes) of scaling operating systems. On one hand, Cerberus incorporates some common wisdom in new operating system designs, such as state replication and message passing. On the other hand, Cerberus is designed based on reusing commodity operating systems, which means Cerberus may still share the benefits of improvements to commodity operating systems. It should be noted that Cerberus also comes at the cost of increased resource consumption due to the increased number of OS instances. However, for future many-core platforms with likely abundant resources, we believe it is worthwhile to trade resources for scalability.

In general, Cerberus could mitigate or avoid many instances of resource contention within a single operating system as well as in the VMM, due to the reduced number of CPU cores managed by a single operating system kernel. It is also easier for the inter-OS communication protocol to scale with the number of OS instances, rather than with the number of cores. Thus, contention within many subsystems could be mitigated for shared-memory multi-threaded and multiprocessing applications.

As well as state replication and distribution in operating systems and the VMM, Cerberus also retrofits some techniques in new OS designs back to commodity operating systems. Cerberus extends traditional system virtualization techniques with support for efficient resource sharing among the clustered operating systems. Specifically, the VMM is built with the *address range* support from Corey [Boyd-Wickizer 2008] to minimize the page fault costs for cross-OS memory sharing, which is critical for some memory-intensive applications. Further, to reduce contention for file accesses, Cerberus incorporates an efficient distributed file-system among clustered OSes, which optimizes local accesses while maintaining good performance for remote accesses.

Moreover, Cerberus incorporates a system call virtualization layer that allows processes/threads of an application to be executed in multiple operating systems, yet provides users with the illusion of running in a single operating system. This layer relies on both message passing and shared memory mechanisms to route system calls to specific operating systems and marshal the results, thus providing applications with a unified TCP/IP stack and file system. This layer uses the notion of "*SuperProcess*", which groups processes/threads in multiple operating systems, to manage the spawned processes/threads.

We have implemented a Cerberus prototype based on Xen-3.3.0 [Barham 2003] and Linux-2.6.18, which runs on an Intel machine with 16 cores and an AMD machine with 48 cores. The prototype adds 1800 lines of code to the Xen VMM and requires no code change to the Linux core kernel. A loadable kernel module and a user-level module are implemented to support the system call virtualization, which has 8,800 lines of code in total.

To measure the effectiveness of Cerberus, we have conducted several performance measurements and compared the performance of a shared memory MapReduce application, dbench [Tridgell 2010], Memcached and Apache web sever running on a single Linux (native Linux and virtualized Linux) and Cerberus with different number of VMs. Performance results show that though Cerberus incurs overhead for some primitives, it does provide better performance scalability. The performance speedup ranges from 1.74X to 4.95X over native Linux and from 1.37X to 11.62X compared to virtualized Linux on 48 cores. The profiling results using Oprofile [Levon 2004] and Xenprof [Menon 2005] indicate that Cerberus mitigates or avoids many instances of contention within both Xen and Linux.

In summary, the contributions of this paper are:

- A technique called OS clustering, which provides a backward-compatible way to scale existing shared memory applications on multicore machines;
- A set of mechanisms to enable efficient sharing of resources among clustered operating systems;
- The design and implementation of our prototype system Cerberus, as well as the evaluation of Cerberus using realistic application benchmarks, which demonstrate both the performance and scalability of our approach.

The rest of the paper is organized as follows. The next section relates Cerberus with previous work on OS scalability. Section 3 provides an overview on the challenges and approaches of Cerberus. Sections 4 and 5 present the design of the two major enabling parts of Cerberus, namely *SuperProcess* and resource sharing. Then, section 6 describes the implementation details on Xen and Linux. The experimental results are shown in section 7. We present the discussion of the limitations and future work in section 8. Finally, we end this paper with a concluding remark in section 9.

2. Related Work

Improving the scalability of UNIX-like operating systems has been a longstanding goal of system researchers. This section relates Cerberus to other work in operating system scalability.

2.1 OS Structuring Strategies

Cerberus is influenced by much existing work on system virtualization, building new scalable OSes and refining existing OSes. Cerberus differs from existing work mainly in that it aims at improving performance scalability of existing applications by using a backward-compatible technique called OS clustering.

The idea of running multiple operating systems in a single machine is not new, but rather an inherent goal of system virtualization [Goldberg 1974]. For example, Disco [Bugnion 1997] (and its relative Cellular Disco [Govil 1999]) had run multiple virtual machines in the form of a virtual cluster to support distributed applications. Denali [Whitaker 2002] also safely multiplexes a large number of Internet services atop a lightweight virtual machine monitor. Cerberus puts these ideas into the context of multicore architecture, and more importantly supports efficiently running a contemporary shared-memory application with POSIX APIs on multiple clustered operating systems with little or no modification.

One viable way to scale operating systems is partitioning a hardware platform as a distributed system and distributing replicated kernels among partitioned hardware. Hive [Chapin 1995] uses a strategy called multicellular, which organizes an operating system as multiple independent kernels (i.e., cells), which communicates with each other for resource management to provide better reliability and scalability. Barrelfish [Baumann 2009] tries to scale applications on multicore system by using a multikernel model, which distributes replicated kernels on multiple cores and uses message-passing instead of shared-memory to maintain their consistency. The factored operating system [Wentzlaff 2008] argues that with the likely abundant cores, it would be more appropriate to space-multiplex cores instead of time-slicing them. Helios [Nightingale 2009] is an operating system that aims at bridging the heterogeneity of different processing units in a platform by using a satellite kernel, which provides the same abstractions across different processor architectures. Cerberus is also influenced by these systems in the use of replicated kernels and state, but retrofits the ideas to commodity operating systems to scale existing shared-memory applications.

Other work has focused on improving OS scalability by controlling or reducing sharing and improving data locality. Corey [Boyd-Wickizer 2008] is an exokernel [Engler 1995] style operating system that provides three new abstractions (share, address range and kernel core) for applications to explicitly control sharing of resources. K42 [Appavoo 2007] and its relatives (Tornado [Gamsa 1999] and Hurricane [Unrau 1995]) are designed to reduce contention and to improve locality for NUMA systems. Cerberus shares some similarities with the clustered objects of K42, but applies at a much higher level (complete operating systems).

2.2 Efforts in Commodity OSes

There are extensive studies on the scalability issues of commercial kernels and a lot of approaches are proposed to fix them. RCU [McKenney 2002], MCS lock [Mellor-Crummey 1991] and local runqueues [Aas 2005] are strategies that aim at reducing the contention on shared data structures. Recently, Boyd-Wickizer et al. [Boyd-Wickizer 2010] analyzed and fixed the scalability of many-core applications on Linux by refining the kernel, and improving applications' user-level design and use of kernel services. Cerberus also aims at improving scalability of applications on Linux, but runs multiple commodity OS instances to host one application instead of refining the core kernel.

In summary, the effort of Cerberus is complementary to the efforts of improving the scalability of existing commercial operating systems. With more scalable operating systems, Cerberus would require fewer operating systems to be clustered together to provide a scalable runtime environment.

3. Overview and Approaches

This section first discusses the challenges and illustrates the solutions to running a shared-memory application on multiple operating systems in a virtualized system. Then, we give an overview of the Cerberus architecture.

3.1 The Case for OS Clustering

To meet its design goals, Cerberus uses pervasive system virtualization. Rather than designing a new OS from scratch or fixing the internal mechanisms of commodity OSes, Cerberus clusters multiple commodity operating systems atop a VMM, and allows an application to run on multiple clustered operating systems with a shared-memory interface. Using multiple federated operating systems hosted by a VMM to run one application means that processes/threads belonging to one shared-memory application now run on multiple OSes. Hence, Cerberus uses a set of mechanisms to ensure system consistency.

Single Shared-Memory Interface: To avoid requiring a port of existing applications, it is critical to provide the existing shared-memory interface to applications. Usually, application programmers using traditional shared-memory APIs (e.g., POSIX) often make the assumption that their programs run within an operating system. Thus, a shared-memory application running in an operating system in the form of multiple processes and threads often expects to share a consistent view of system resources. These processes/threads also rely on the operating system interfaces and services to communicate with each other. For example, threads belonging to one process are expected to see the same address space and processes in one application have parent-children relations and use IPCs to notify each other.

To address these issues, Cerberus incorporates a system-call virtualization layer, which coordinates system calls in multiple clustered operating systems and marshals the results. Cerberus uses the notion of *SuperProcess* (section 4), which denotes a group of processes/threads executing in multiple operating systems. Each process/thread in a *SuperProcess* is called a *SubProcess*. The *SuperProcess* coordinates the control delivery and data communication of its *SubProcess*es, to maintain the system's consistency, and provide

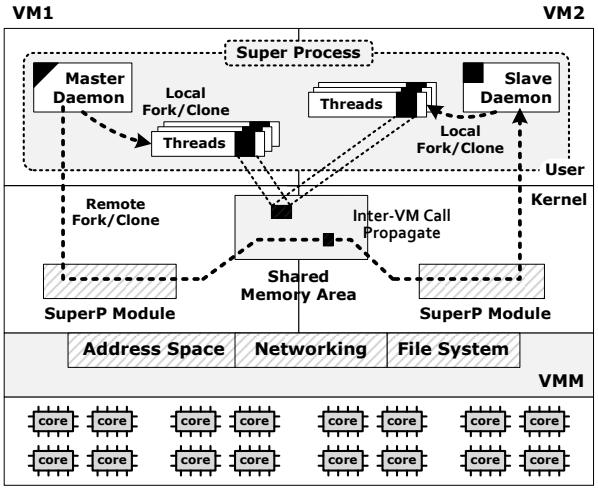

Figure 2. *The Cerberus System Architecture. Cerberus is composed of an extension in the VMM, which handles the resource sharing; and a kernel module in the guest OS, which manages the* SuperProcess *and inter-VM system calls.*

the application with the illusion of running on one operating system.

Efficient Resource Sharing: Another challenge is efficient sharing of resources among processes/threads crossing the operating system boundary, to still provide applications with a consistent view of system resources. Unfortunately, traditional VMMs are not built with support for sharing many resources between operating systems, but rather, enforce strong isolation among guest operating systems for security reasons.

Hence, Cerberus implements a resource-sharing layer in both the VMM and OSes, which supports efficient sharing of resources such as address spaces, networking and file systems. The resource-sharing layer exploits the fact that the clustered OSes share the hardware to coordinate accesses to shared resources. Cerberus uses both shared-memory and message-passing mechanisms to coordinate accesses and events among clustered operating systems. To serialize accesses to resources shared by multiple OS instances, it uses message-passing and lock-free mechanisms when necessary. For events among instances of different operating systems, Cerberus uses a two-level message queue to deliver these events. As the system state of an application is replicated and by default private, false sharing and unnecessary serialization can be significantly avoided.

3.2 System Architecture

The system architecture of Cerberus is shown in Figure 2. There are several virtual machines running atop a VMM. The VMM manages the underlying hardware resources and partitions the resources among the VMs. Currently, Cerberus requires the VMs to run the same operating system kernel for simplicity. Roughly speaking, Cerberus organizes the *SuperProcess* in the form of a coordinated distributed system, using both messages and shared memory. Multiple processes of one application run on multiple OSes in the form of a *SuperProcess*, which consists of one master daemon and multiple slave daemons. There is exactly one slave daemon for an application in VMs not running the master daemon. The master daemon is responsible for loading the initial parts of an application and creating the slave daemons. Afterwards, the master daemon works similarly to the slave daemon, according to the semantics of the application.

The daemons communicate with each other to decide which VMs should serve a process/thread creation request, to balance load among clustered OSes. To run a process in a VM other than the requesting VM, the *SuperProcess* daemon issues a *remote spawn*, which replicates the current running state to the target VM.

Cerberus also routes system calls using the SuperProcess *module* in each operating system, which is a loadable kernel module. The module intercepts system calls made by an application. To retain the semantics of system calls and a consistent view of the execution context, the module routes the system calls, as well as marshalling and translating the results.

Cerberus uses cross-VM message-passing mechanisms to handle communication between daemons in multiple VMs. A daemon uses the message-passing mechanism to send process/thread creation requests and signal remote processes/threads. There is also a shared-memory area for data communication among multiple VMs.

Currently, Cerberus decides the number of operating systems to run based on a user-specified heuristic for simplicity (the scalability limit of an application with certain number of cores). By default, Cerberus allocates a fixed, equal portion of resources to an operating system and lets the application decide the assignment of processes/threads to operating systems. Each operating system is pinned on a fixed number of cores.

In the following sections, we will describe the mechanisms in Cerberus to support *SuperProcess* (section 4) and efficient sharing of address space, file system and networking among clustered operating systems (section 5).

4. Supporting SuperProcess

This section describes the underlying design to support the *SuperProcess* abstraction, which provides applications with the illusion of running on a single operating system.

4.1 Remote Process/Thread Spawning

Cerberus uses techniques from traditional process checkpoint/restart mechanisms to support remote process spawning. As shown in Figure 3, Cerberus first checkpoints the state of the current running process, including the register state, memory mappings and opened files, among others. The checkpointed process state is then put into a shared

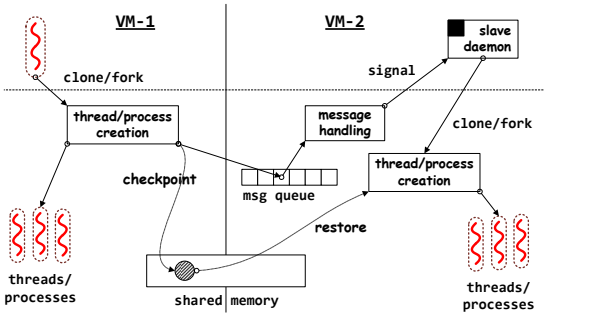

Figure 3. *The sequence of doing a remote fork/clone.*

memory area. To spawn a process on behalf of the requesting daemon in a different operating system, the daemon in the target OS first spawns a child process itself. Then, it retrieves the checkpointed state from the shared memory area, and restores it to the child process.

Currently, many applications use the threaded programming model. As all threads of a threaded application share the same binary image, Cerberus proactively creates a resident process (similar to the dispatcher in K42 [Krieger 2006]) for each clustered operating system, and maintains the consistency of each resident process by propagating changes to the application's global resources. For example, Cerberus automatically propagates memory mapping and unmapping requests in the issuing operating system to other clustered operating systems. Thus, for applications that create a large number of threads, the cost of remote spawning of threads is reduced, as the thread creation requests can be done locally by each resident process.

It should be noted that although creating a remote process or thread in Cerberus is more heavyweight than within a single OS, Cerberus supports parallel fork/clone that allows simultaneously creating processes/threads in multiple VMs, which amortizes the cost of a single operation.

4.2 Process Management

Cerberus relies mainly on the system call interception and redirection mechanisms to group processes distributed across multiple operating systems to provide correct semantics.

Cerberus virtualizes the process identity (such as the process ID), the parent-child relationship and the group information. To achieve this, Cerberus intercepts the system calls manipulating such information, translates the arguments before dispatching the operations, and marshals the results before returning to applications.

For process IDs, Cerberus maintains a global mapping table of the virtual process ID (seen by applications) and the physical ID (seen by the operating system). Cerberus thus relies on the virtual ID to maintain the process relationship. For example, the PID passed by the *kill* shell command will be translated by the Cerberus system call interception layer. If the virtual PID belongs to the current operating system, the signal will be delivered to the process associated with the real PID. Otherwise, Cerberus will redirect the signal to the corresponding operating system. Cerberus also maintains a logical to physical CPU mapping table and provides the correct cores and operating systems to run threads and processes. For example, the pthread library provides interfaces to get and set the affinity (pthread_get(/set)_affinity) that obtain the set of cores on which a thread can run and assign specific threads to run on some cores, and Cerberus translates these calls.

4.3 Coordination of State Accesses

As the state of an application is shared or replicated among multiple clustered operating systems, Cerberus uses lock-free mechanisms and message passing to coordinate changes to the state from each OS. For some shared state among OSes, Cerberus uses compare-and-swap to allow each OS to eagerly access some replicated state such as page table pages. Upon a conflict, Cerberus rolls back the changes to state from one OS. For some shared data structures such as the virtual file descriptor table and inode table, Cerberus partitions these data structures to individual OSes, to avoid access serialization and cache ping-ponging, and uses message passing to coordinate the state.

Cerberus also implements an inter-VM notification mechanism that uses a hierarchical message-passing mechanism: when a process notifies processes in other VMs on the occurrences of certain events (e.g., signals, unmap requests), it first sends a message to the *SubProcess* in that VM. The *SubProcess* will queue the message marked with the type of the message and deliver the message to the appropriate VM. Then the receiver VM will send the corresponding event to the appropriate threads/processes. For example, for a futex [Franke 2002][1] call on the address of a remote thread, Cerberus will translate the address into the real address to monitor. On being notified by the local operating system about the change of the address, Cerberus will send a message to the receiving thread.

5. Supporting Resource Sharing

Cerberus supports the efficient sharing of address spaces, file systems and networks across the clustered operating systems, to provide a consistent view for applications.

5.1 Sharing Address Spaces

Cerberus identifies the range of shared address space by interpreting the application's semantics. An application running in a multi-threading mode should normally have its address space shared with all threads in a process. A forked process usually shares little with its parent. For a multi-threaded application, Cerberus maintains a global list of the

[1] A futex allows two entities to synchronize with each other using a shared memory location. The pthread mutex is implemented based on this mechanism.

shared *address ranges*. It intercepts the memory mapping requests (e.g., *mmap*) from each thread and updates the list accordingly. Cerberus creates a virtual memory mapping for that shared address range to let the page fault handler be aware of that address range. When handling a page fault, the Cerberus module first checks for a pending list entry and updates the virtual memory mapping before resolving the faulting address.

To efficiently share an address space across operating systems, Cerberus incorporates the *address range* abstraction from Corey [Boyd-Wickizer 2008]. This supports sharing a subset of the root page table by multiple guest VMs, according to the address range. The level of page table sharing might be changed according to the virtual memory mapping of an operating system. Cerberus also dynamically coalesces and splits the sharing of page tables according to the application's memory mapping requests. According to the list of shared address ranges, the page fault handler in the VMM will connect the page table of a shared address range in one VM's page table to that in other VMs, when there is a first access to that shared address range in those domains. Cerberus determines the level of sharing based on the size of the address range.

5.2 Sharing File Systems

Running a single application on multiple operating systems raises the problem of sharing files among processes in each clustered operating system. This is because each operating system will have its own file system and device driver, preventing a process from accessing files managed by another operating system. One intuitive approach would be the use of a networked file system managed by one operating system, with other operating systems as NFS clients to access files in the operating system running the NFS server. However, this creates two performance problems. First, all file accesses are now centralized to one operating system, which can easily make the accesses the new performance bottleneck. Second, there are some inherent performance overheads, as a networked file system usually has inferior performance compared to a local one. For example, recent measurements [Nightingale 2005, Zhao 2006] showed that NFS could be several times slower than a native file system such as ext3.

Fortunately, most files in many multiprocessing applications are usually accessed exclusively, with few opportunities to be accessed by multiple processes (except some non-performance-critical ones such as log files)[2]. Hence, Cerberus uses a hybrid approach of both networked and local file system, which seeks to give accesses to private files little contention and high performance, while maintaining acceptable performance for shared files.

[2] For multi-threaded applications, applications usually map the files into memory using mmap and then threads can modify the memory-mapped file directly, which will be discussed in the following sections.

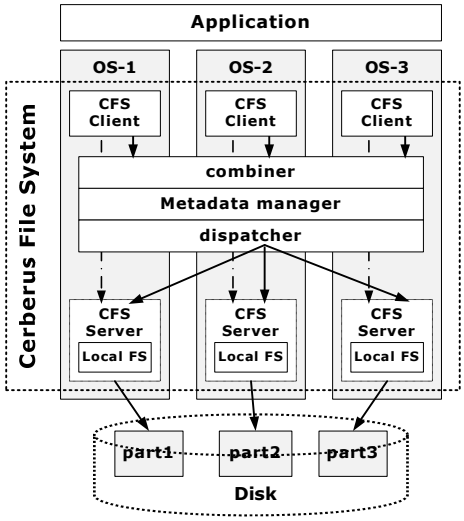

Figure 4. *Architecture of the Cerberus file system (CFS), which is organized as a mesh of networked file systems: each OS manages its local partition and exposes it to other OSes through the CFS client. Cerberus dispatches accesses to files and marshals the results according the managed metadata.*

Figure 4 shows the architecture of our approach, which forms of a mesh of networked file systems: each operating system manages a local partition and exposes it to other operating systems through an NFS-like interface; processes in such an operating system access private files directly in the local partition and access files in other partitions through the CFS client. To identify a file as shared or private, Cerberus maintains a mapping from each inode describing a file to the owner ID (e.g., virtual machine ID). As the metadata of files in each partition is maintained only by one operating system, Cerberus offers a similar metadata consistency and crash-recover model to native systems. It should be noted that the CFS implemented by Cerberus does not rely on network but rather on the virtual machine communication interfaces for communication and shared memory. This avoids redundant file data copies and the associated data exchange, and thus is more efficient than NFS [Zhao 2006]. Again, the sharing of a file between the CFS client and CFS server is done using the *address range* abstraction to minimize soft page faults.

To provide applications with a consistent view of the clustered file system, Cerberus intercepts accesses to the attributes or state of each file and directory, distributes accesses to each partition when necessary, and marshals the results before returning to user applications. Such operations (e.g., list a directory) are relatively costly compared to those in a single operating system. However, they are rare and usually occur in non-performance critical paths of applications.

5.3 Sharing File Contents

For multithreaded applications, it is common that the content of a file is shared by multiple threads. Thus, Cerberus sup-

ports the sharing of a file based on the *address space sharing* in Cerberus to maintain consistency for a file accessed by multiple operating systems. Cerberus uses memory mapped I/O (MMIO) to map a file into a shared address range, which is visible to all threads in multiple operating systems. Cerberus only allows the *SuperProcess* to access shared files using MMIO. To provide backward compatibility for applications using the traditional read/write APIs, Cerberus handles file I/O to shared files using a similar idea to that in Overshadow [Chen 2008], by translating file related I/Os to MMIOs. On the first read/write operation to the file, Cerberus maps the file in a shared address space using the *mmap* system call. Cerberus ensures that the buffer is mapped using the *MAP_SHARED* flag. Cerberus also ensures that the address range of the memory buffer is shared among clustered operating systems using the *address range* abstraction. Thus, changes from one operating system will be directly visible to other operating systems. Then, Cerberus emulates the read/write system calls by operating on the mmapped area.

To provide file-I/O semantics, Cerberus maintains a virtual file metadata structure that reflects the logical view of the files seen by a process. Cerberus also virtualizes the system calls that operate on the metadata of files. For example, the fseek system call will advance the file position maintained in the virtualized metadata and return the state in virtualized metadata for fstat-like system calls.

Note that this scheme is transparent to the in-kernel file systems and buffer cache management, as each buffer cache will have a consistent view of the file. The same piece of a file might be replicated among multiple buffer caches, causing wasted memory. However, multiple replicas also increase the concurrency of file access and avoid unnecessary contention.

5.4 Shared Networking Interfaces

To provide applications with a consistent view of networking interfaces, Cerberus exploits the fact that typical servers are usually equipped with multiple NICs, and each NIC is now hardware virtualizable (e.g., through SR-IOV [PCI-SIG 2010]). Hence, Cerberus directly assigns either virtualized NICs or physical NICs to each operating system for high performance. This could avoid contention on TCP/IP stacks if the operations are done on the local (virtual) NICs. To hide applications from such geographic distributions, Cerberus virtualizes the socket interface by intercepting related system calls and relies on the file descriptor virtualization described previously to manage socket descriptors. Cerberus maintains the (virtual) NIC information, and redirects calls that bind to a NIC if necessary. Cerberus then dispatches related operations (e.g., send, receive) to the VM that manages the NIC. The associated data will be exchanged using the shared memory area managed by Cerberus to avoid possible data copies.

6. Prototype Implementation

We have implemented Cerberus based on Xen to run multiple Linux instances with a single shared memory interface, using the shadow mode of page table management in Xen. The system call layer in Cerberus currently supports only a subset of the POSIX interface, but is sufficient to run many applications including shared-memory MapReduce applications, Apache, Memcached and file system benchmarks. For simplicity, Cerberus currently requires applications to be statically linked[3], and to link with a small piece of user-level code containing a few Cerberus-specific signal handlers, that handle remote requests such as futex and socket operations.

6.1 Inter-VM Message Passing

The inter-VM message passing mechanism is implemented by leveraging the cross-VM event channel mechanism in Xen. Cerberus creates a point-to-point event channel between each pair of clustered operating systems. The *SuperP* module inside each operating system has a handler to receive such cross-VM events and distribute them to the receivers. In the case of concurrent cross-VM events, each operating system maintains a cross-event queue to buffer the incoming events, and handles them in order. All cross-VM communication of Cerberus, such as futex and signal operations, uses this mechanism.

6.2 Memory Management

In Cerberus, the sharing of page tables is implemented in the shadow page tables, and by manipulating the P2M (physical-to-machine) table, thus is transparent to guest operating systems. We have also investigated an implementation of page sharing for Xen's direct mode (with writable page tables), with the aim of supporting para-virtualization. However, our preliminary results show that supporting writable page tables could result in significant changes to guest operating systems, as well as incurring non-trivial performance overhead.

On x86-64, Xen uses 4-levels of page tables and Cerberus supports sharing at the lower three levels (i.e., L1 – L3). Cerberus records the root page table page for an address range when the guest kernel connects an allocated page table page to the upper-level page table. When sharing a page table page among multiple OSes, one machine page might be accessed by multiple OSes, and thus might correspond to more than one guest-physical page in Xen. Hence, Cerberus creates a per-VM representation of each shared page table, but in an on-demand way. When a VM tries to write a page table page for the first time, Cerberus will create a representation of the page table page in that VM and map it to a single machine page by manipulating the P2M table, which maps guest physical memory to the host machine memory. Cerberus uses compare-and-swap to serialize updates to shared

[3] This will not increase much memory usage, as application code is shared by default.

page table pages among multiple VMs: when a VM tries to update the shared page table, it uses a compare-and-swap to see if the entry has already been filled by other VMs, and frees the duplicated page table page if so.

Other than sharing page tables, Cerberus also needs to synchronize the virtual memory area (VMA) mappings across clustered VMs. As threads on different VMs have separate address spaces, they maintain their VMAs individually. Memory management system calls (e.g., mmap) on a single VM only change the VMA mappings of the threads in that VM. Thus, Cerberus intercepts most memory management system calls (e.g., mmap, mremap, mprotect, munmap and brk). Before handling the memory management system call, Cerberus will first force the VM to handle the virtual memory synchronization requests from other VMs. After finishing the call, Cerberus will allow the VM to propagate the system call to all other VMs in the system. This is done by adding a virtual memory synchronization request with appropriate parameters to the request queue of each receiver VM.

6.3 Cerberus File System

Inodes in a Cerberus file system (CFS) are divided into two kinds, namely local inodes and remote inodes. Local inodes describe files on a domain-local file system, and may be accessed directly. Remote inodes correspond to files stored on remote domains. A remote inode can be uniquely identified by its owner domain and its inode number in that domain. When a remote inode is created, CFS will keep track of this unique identifier. Each time a remote inode access is required, CFS will pack the inode identifier and other information into a message, and send it to remote domain via the inter-VM message passing mechanism.

Another data structure we track is the dentry. A dentry is an object describing relationships between inodes, and storing names of inodes. Unlike inodes, dentries in the original Linux file system do not have identifiers. To simplify remote dentry access, we assign a global identifier to each remote dentry. The dentry id is assigned in a lazy way, that is, only when a dentry is visited from a remote domain for the first time, will we assign a global identifier to it.

6.4 Virtualizing Networking

Cerberus virtualizes the socket interface by intercepting the related system calls. The socket operations are divided into two kinds, namely local and remote socket operations. We use virtual file descriptor numbers to distinguish the operations. Each virtual file descriptor number is associated with a virtual file descriptor. The virtual file descriptor describes the owner VM, the *responder* (a user-level daemon on the owner VM) and the real file descriptor corresponding to it. When a process accesses a virtual file descriptor, Cerberus will first check the corresponding owner VM. If it is a local access, Cerberus just handles the request as in native XenLinux using the real file descriptor. Otherwise, Cerberus will send a remote socket operation request to the target VM, and let the *responder* handle the socket request. With this simple mechanism, Cerberus can currently support several socket-related operations (such as bind, listen, accept, read, write, select, sendto and recvmsg).

6.5 System Call Virtualization

We classify system calls into two types according to which system state they access. The first type includes system calls that only access local state or are stateless (e.g., get_systime). For such system calls, replicating calls among multiple OSes will not cause state consistency problems, and thus Cerberus does not need to handle them specially. The second type includes system calls that access and modify global state in the operating system (e.g., mmap). Cerberus needs to intercept this kind of system call, coordinate state changes, and marshal the results to support cross-VM interactions. To do interceptions, the Cerberus module modifies the system call table to change the function pointers of certain system call handlers to Cerberus-specific handlers during loading. When a system call is invoked, the Cerberus handler checks if it should be handled by Cerberus, and if so, invokes specific handlers provided by Cerberus.

We have currently virtualized 35 POSIX system calls (belonging to the second type) at either system call level or virtual file system level. They are divided into five categories: process/thread creation and exit (e.g., fork, clone, exec, exit, exit_group, getpid and getppid); thread communication (e.g., futex and signal); memory management (e.g., brk, mmap, munmap, mprotect and mremap); network operations (e.g., socket, connect, bind, accept, listen, select, sendto, recvfrom, shutdown and close); and file operations (e.g., open, read, write, mkdir, rmdir, close and readdir). We currently leave system calls related to security, realtime signals, debugging and kernel modules unhandled. In our experience, virtualizing a system call is usually not very difficult, as it mostly involves partitioning/marshaling the associated cross-process state. Table 1 gives some typical examples of how they are implemented.

6.6 Implementation Efforts

In total, the implementation adds 1,800 lines of code to Xen to support management of Cerberus and efficient sharing of data among *SubProcess* in multiple Linux instances. The support for system call interception, super-process and Cerberus file system is implemented as a loadable kernel module, which is comprised of 8,800 lines of code. It takes 1,250 lines of code to enable *SuperProcess* management. About 800 lines of code are used to support network virtualization and 750 lines of code to support the Cerberus file system. The Cerberus system call virtualization layer takes about 3000 lines of code, including marshaling multiple system calls (e.g., clone). The Cerberus system support code consists of 3000 lines, including the management of shared memory pool, cross-VM messages and process

Syscall	Approaches
clone	Cerberus first makes sure each VM has the *resident process*. Then, it queries the *SuperProcess* daemons for the target domain. Native clone is invoked for a local clone. Otherwise a remote clone request with the marshalled parameters (e.g., stack address) is sent. The resident process on the target domain then creates a new thread.
getpid	Cerberus returns a virtual pid to the caller. The virtual pid contains the domain id and the *SubProcess* number.
signal	Cerberus scans the mapping between virtual pid and process to find the target domain and process. A remote signal request is sent when necessary. The native signal call is then invoked on the receiver domain.
mmap	Cerberus first handles the VMA synchronization request, and then makes the native mmap call. Finally, it broadcasts the mmap result to other VMs.
accept	Cerberus checks the virtual fd table to get the owner domain of the fd. A remote accept request is sent when necessary. The accept operation is done by the corresponding *responder* with the real fd, and the resulting virtual fd of the created connection is sent back.
sendto	If the fd does not refer to a remote connection (either the socket is not established or it is a local connection), Cerberus will invoke the native sendto. Otherwise, Cerberus will query the virtual fd table to get the owner domain. A remote sendto request is sent and handled by the corresponding *responder* with the real fd.
mkdir	Cerberus first gets the global identifiers of the inode and dentry of the parent directory. If it is a local request, Cerberus passes it to the native file system. Otherwise, Cerberus gets the owner domain id and sends a remote CFS request with the global identifiers, the type of the new node (directory in this case) and the directory name. The owner domain creates the new child directory and sends the corresponding global identifier of the newly created inode and dentry back to the request domain.

Table 1. *System call implementation examples*

checkpointing and restoring (including 700 lines of code from Crak [Zhong 2001]).

7. Experimental Results

This section evaluates the potential costs and benefits in performance and scalability of Cerberus's approach to mitigating contention in operating system kernels.

7.1 Experimental Setup

The benchmarks used include histogram from the Phoenix testsuite [Ranger 2007][4], dbench 3.0.4 [Tridgell 2010], Apache web server 2.2.15 [Fielding 2002] and Memcached 1.4.5 [Fitzpatrick 2004].

Moreover, we present the costs of basic operations in Cerberus. We use OProfile to study the time distribution of *histogram*, *dbench*, *Apache* and *Memcached* on Cerberus, Xen-Linux and Linux.

Most experiments were conducted on an AMD 48-core machine with 8 6-core AMD 2.4 GHz Opteron chips. Each core has a separate 128 KByte L1 cache and a separate 512 KByte L2 cache. Each chip has a shared 8-way 6 MByte L3 cache. The size of physical memory is 128 GByte. We use Debian GNU/Linux 5.0, which is installed on a 147 GByte SCSI hard disk with the ext3 file system. There are a total of four network interface cards and each is configured with different IPs in a subnet. The input files and executable for testing are stored in a separate 300 GByte SCSI hard disk with ext3 file system. The Apache and Memcached benchmarks were conducted on a 4 quad-core Intel machine with 8 NICs (as it has more NICs than the AMD machine), to reduce the bottlenecks from the NIC itself. Due to resource limitations, we can run up to 24 virtual machines on the AMD machine. All performance measurements were tested at least three times and we report the mean.

Cerberus is based on is Xen 3.3.0, which by default runs with the Linux kernel version 2.6.18. We thus use the kernel version 2.6.18 for the three measured systems. Xen-Linux uses the privileged domain (Dom0) in *direct paging mode*, for good performance. As Xen-3.3.0 can support at most 32 VCPUs for one VM, we only evaluate Xen-Linux with up to 32 cores.

We compare the performance and scalability of Cerberus with Linux. We also present the performance results of Xen-Linux to show the performance overhead incurred by the virtualization layer, as well as the performance benefit of Cerberus over typical virtualized systems. To investigate the performance gain of Cerberus, we used Oprofile and Xenoprof to collect the distribution of time of *histogram*, *dbench*, *Apache* and *Memcached* on Xen-Linux and Linux and that on Cerberus using the 2 core per-VM configuration. All profiling tests use the *CPU_CYCLE_UNHALTED* event as the performance counter event.

7.2 Cerberus Primitives

We also wrote a set of microbenchmarks to evaluate the cost of many primitives of Cerberus, to understand the basic cost underlying Cerberus.

Sending Signals: To evaluate the performance of the Cerberus signal mechanism, we use a micro-benchmark to test the time it takes to send a signal using a ping-pong scheme (e.g., sending a signal to a process and that process sending a signal back to the originator) on both the Intel and AMD machines. Table 3 depicts the evaluation results. It can be seen that the virtualization layer introduces some overhead

[4] The reason we chose histogram is because it has severe performance scalability problems on our testing machine, which other programs don't exhibit.

	localhost	remote host
Native Linux	12.5ms	125.5ms
Xen-Linux	42.9ms	132.6ms
Cerberus local	43.1ms	131.8ms
Cerberus remote	87.1ms	154.7ms

Table 2. *Cost of ping-ponging one packet 1000 times*

	Intel	AMD
Native Linux	7.9ms	4.0ms
Xen-Linux	38.7ms	74.1ms
Cerberus local	43.1ms	72.3ms
Cerberus remote	25.8ms	45.0ms

Table 3. *Cost of ping-ponging 1000 signals*

to the signal mechanism. However, sending a cross-VM signal takes less time than sending a local signal. There are two reasons: 1) The inter-VM message passing mechanism is efficient; 2) Sending a signal to a remote process only needs to forward the request to the target VM, so signaling the target process and executing the sender process can be done in parallel.

Primitive	Config	Time
remote fork	1 process	5.40 ms
	24 processes	31.77 ms
remote clone	1 thread	3.21 ms
	24 threads	30.79 ms

Table 4. *The costs of fork and clone in Cerberus*

Remote Fork and Clone: The first and second columns of Table 4 show the cost of spawning 1 processe/thread on a remote VM in the AMD machine with 2 VMs and concurrently spawning 24 processes/threads on remote VMs in the AMD machine with 24 VMs. Cerberus suffers from some overhead due to checkpointing, transferring and restoring process/thread state from the issuing VM to the receiving VMs. However, with increasing numbers of VMs, the steps of creating remote threads can be processed in parallel. This helps to reduce some overhead of creating threads as shown in the table.

Inter-VM Message Passing: To evaluate the costs of inter-VM message passing, we pass a message between VMs in order using a ping-pong scheme, e.g., sending that message to a VM and the VM responds by sending a message back to the sender. The time for one round-trip is around 10.24 μs within the same chip and 11.34 μs between chips, which we believe is modest and acceptable.

Reading a File with CFS: To evaluate the performance of (CFS), we write a micro-benchmark to test the time it costs to read the beginning portion of a simple file on the AMD machine. We generate one hundred files with random content, clear the buffer cache, read the first ten bytes of each file, and then calculate the average execution time. The result shows that one read operation on a native Xen-Linux file takes 6.47 μs, and one read operation on a local CFS file takes 6.52 μs, while on a remote CFS file it will cost 17.81 μs. The performance of local read operations on the CFS is close to that of native Xen-Linux file system. However, remote read operations introduce some performance overhead.

Sending and Receiving Packets: To evaluate the performance of the Cerberus network system, we use a micro-benchmark to test the time for sending and receiving network packets, using a ping-pong scheme on the Intel machine. The micro-server establishes a network connection with the client and creates a child to handle the following requests. The client will send an 8 byte string to the server through a socket connection (localhost/remote host) to trigger the test. Table 2 depicts the evaluation results. It shows the execution time of ping-ponging one message 1000 times under different configurations. It can be seen that the virtualization layer introduces some overhead for sending and receiving packets, while forwarding a packet in Cerberus introduces more overhead. However, if the connection is from a remote host, the overhead of the packet forwarding is below 25% compared to native Linux.

7.3 Performance Results

For *histogram* and *dbench*, we ran each workload on the AMD 48-core machine under Xen-Linux and Linux with the number of cores increasing from 2 to 48. For Cerberus, we evaluate two configurations, which run one and two cores for each VM (Cerberus-1core and Cerberus-2cores), running on a different number of cores, increasing from 2 to 48. When running one virtual machine on two cores, we configured each VM with cores that have minimal communication costs (e.g., sharing the L3 cache). For the Apache web server and Memcached service benchmarks, we run each workload on the Intel 16-core machine under Cerberus, Xen-Linux and Linux with different number of cores, increasing from 2 to 16. As both applications require a relatively large number of NICs, we did not test them on 48-core AMD machine. During the Apache and Memcached tests, we setup one instance of the web server on each core, which accepts service requests from clients running on a pool of 16 dual-core machines (32 clients for Apache and 64 clients for Memcached).

Histogram: Figure 5 shows the performance and scalability of histogram processing 4 GByte of data on Cerberus, native Linux and Xen-Linux. All input data is held in an in-memory tmpfs to avoid applications being bottlenecked by disk I/O. Cerberus performs significantly worse than Linux for a small number of cores, due to the performance overhead in shadow page management and the inherent virtualization overhead. However, as the number of cores increases, the execution time of histogram eventually decreases and outperforms native Linux. The speedup of Cerberus over Xen-Linux is around 51% on 24 cores for the one core per-

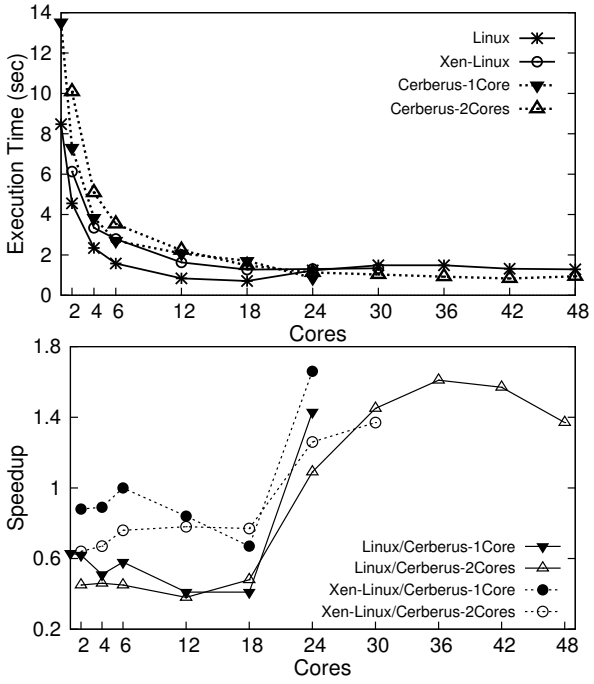

Figure 5. *The execution time and speedup of histogram on Cerberus compared to those on Linux and Xen-Linux under two configurations: which use 1 and 2 cores/domain accordingly.*

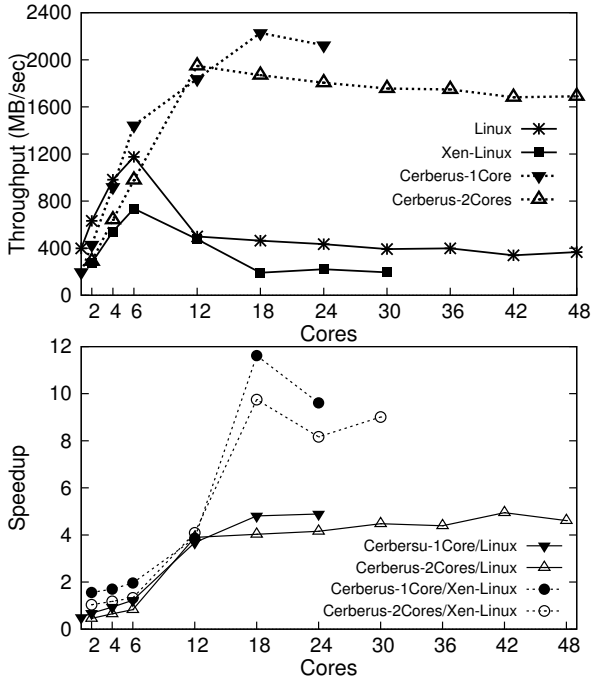

Figure 6. *The throughput and speedup of dbench on Cerberus compared to those on Linux and Xen-Linux under two configurations: which use 1 and 2 cores/domain accordingly.*

VM configuration, and 30% on 30 cores for the two cores per-VM configuration. The speedup over Linux is around 43% on 24 cores for one core per-VM, and 37% on 48 cores for two cores per-VM. The performance of two cores per-VM is worse than that of one core per-VM, due to the increased contention on the shadow page table inside Xen. The speedup of Cerberus degrades a little (57% vs. 37%) from 42 cores to 48 cores, probably because the costs of creating threads and communication increases, thus the benefit degrades.

Table 5 shows the top 3 hottest functions in the profiling report of the *histogram* benchmark. Linux suffers from contention in __up_read and __down_read_trylock due to memory management. Xen-Linux spends most of its time in address 0x0 (/vmlinux-unknown) when the number of cores exceeds eight[5], which might be used for a para-virtualized kernel to interact with the hypervisor. However, Cerberus does not encounter contention in Linux and Xen-Linux, the time spent in the lock-free implementation (cmpxchg) increases a little with the increasing number of cores.

dbench: Figure 6 depicts the throughput and speedup of dbench on Cerberus over Xen-Linux and Linux. The throughput of dbench on Xen-Linux and Linux degrades dramatically when the number of cores increases from 6 to 12 and degrades slightly afterwards. By contrast, though the throughput of Cerberus is worse than that on Linux for a small number of cores (1-6), its throughput scales well to 18 cores and 12 cores for the one and two core per-VM configuration. It appears that dbench has reached its extreme throughput here and has no further space for improvement. Starting from 12 cores or 18 cores, the throughput degrades slightly due to the increased process creation and inter-VM communication costs. Again, the one core per-VM configuration is slightly better than the two cores per-VM configuration, due to the per-VM lock on the shadow page table. In total, the speedup is 4.89X for the one core per-VM configuration on 24 cores, 4.95X for 42 cores, and 4.61X for 48 cores.

Table 6 shows the top 3 hottest functions in the profiling report of *dbench* benchmark. We ignore the portion of samples related to *mwait_idle*, as it means the CPU has nothing to do. From the table we can see that Linux and Xen-Linux both spend substantial time in ext3 file system operations, which may be the reason for poor scalability. On the other hand, Cerberus does not encounter such scalability problems, but is slightly affected by the shadow paging mode.

The evaluation on histogram and dbench also shows that these applications poorly utilize multicore resources when the number of cores reaches a certain level. This indicates that horizontally allocating more cores to such applications may not be a good idea. Instead, allocating a suitable amount of cores to such applications could result in better utilization and performance tradeoff.

[5] The profiling results are obtained through Xenoprof using the *CPU_CYCLE_UNHALTED event*

Threads	Top 3 Functions	Percent
Linux		
48	__up_read	38.6%
	__down_read_trylock	35.9%
	calc_hist	8.3%
1	calc_hist	81.2%
	find_busiest_group	0.06%
	page_fault	0.03%
Xen-Linux		
32	/vmlinux-unknown	70.9%
	calc_hist	11.6%
	__handle_mm_fault	3.2%
1	calc_hist	60.3%
	__handle_mm_fault	3.6%
	sh_gva_to_gfn__guest_4	2.7%
Cerberus		
2/VM	calc_hist	22.5%
	sh_x86_emulate_cmpxchg__guest_2	8.9%
	/xen-unknown	8.3%

Table 5. *The summary of the top 3 hottest functions in histogram benchmark profiling*

Threads	Top 3 Functions	Percent
Linux		
48	ext3_test_allocatable	66.6%
	bitmap_search_next_usable_block	18.2%
	journal_dirty_metadata	0.02%
1	/lib/libc-2.7.so	20.7%
	copy_user_generic	14.1%
	__d_lookup	0.03%
Xen-Linux		
32	ext3_test_allocatable	59.7%
	bitmap_search_next_usable_block	17.7%
	/vmlinux-unknown	5.99%
1	/lib/libc-2.7.so	13.7%
	copy_user_generic	9.9%
	__d_lookup	4.1%
Cerberus		
2/VM	sh_x86_emulate_cmpxchg__guest_2	11.2%
	/xen-unknown	8.67%
	sh_x86_emulate_write__guest_2	5.2%

Table 6. *The summary of the top 3 hottest functions in dbench benchmark profiling*

Apache Web Server: Figure 7 shows the per-core throughput of Apache on the Intel 16-core machine under Cerberus, Xen-Linux and Linux. There are a total of eight NICs, and each is configured with a different IP in a subnet. We run one web server instance on each core and share one NIC between two web servers. The throughput of Apache on Linux significantly degrades with the growing number of cores. When evaluating Cerberus, we directly assign 8 NICs

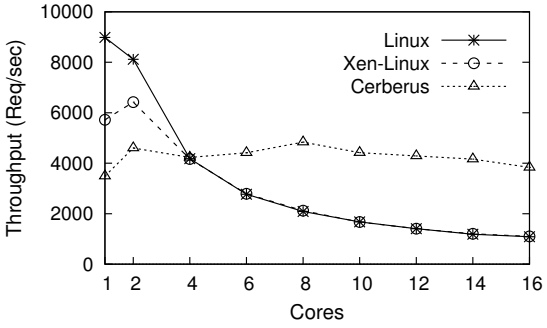

Figure 7. *The per-core throughput of Apache on Cerberus compared to those on Linux and Xen-Linux*

to 8 different VMs (using PCI passthrough). The per-core throughput of 16 cores is only 1085 requests/sec for Linux, which is 12.1% of that on 1 core. By contrast, the throughput of Cerberus is quite stable. Although Cerberus performs worse than Linux for a small number (1-2) of cores (4603 vs. 8118 on 2 cores), it outperforms Linux when the number of cores exceeds 4 and scales nearly linearly. Cerberus achieves a speedup of 3.49X and 3.53X over Linux and Xen-Linux (3833 vs. 1099 and 1085).

The profiling of *Apache* shows that more CPU time is spent idle with the increasing number of cores used to host web servers, and there is some load imbalance. Oprofile shows that the same server instance takes 2.57X more CPU cycles under 1-core configuration than that under 16-core configuration. The same scenario also appears in Xen-Linux(2.39X). This may be caused by contention in the network layer in Linux and Xen-Linux. However, Cerberus does not encounter such a problem and can fully utilize its CPU resources. This evaluation shows that Cerberus could also avoid some imbalance caused by Linux, and achieve more efficient use of resources.

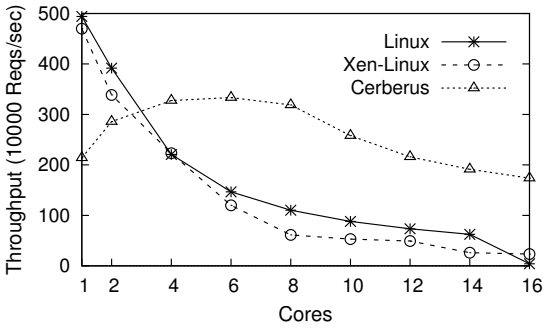

Figure 8. *The per-core throughput of Memcached on Cerberus compared to that on Linux and Xen-Linux*

Memcached: Figure 8 shows the average throughput of Memcached server on the Intel 16-core machine under Cerberus, Xen-Linux and Linux. The configuration is similar to Apache. We run one Memcached server instance on each core and share one card by two servers listening to different UDP ports. The throughput of Memcached server on Linux

significantly degrades when the number of cores exceeds 4. By contrast, the throughput of Cerberus does not degrade until the Memcached instances start to provide service on the same VM, as two instances affect each other heavily. However Cerberus still outperforms Xen-Linux and Linux.

The profiling of *Memcached* shows that many CPU cycles are spent polling network events. Further per-CPU profiling shows that a few Memcached instances spend much time in the *ep_poll_callback* and *task_rq_lock* functions, and seem to block other instances.

We also evaluated *histogram*, *dbench*, *Apache* and *Memcached* on other Linux versions (Linux 2.6.26, the standard kernel for Debian GNU/Linux, and Linux 2.6.35, the newest stable kernel). Only the scalability of *histogram* improves in Linux 2.6.35. Others still suffer from heavy contention, and have similar performance and scalability.

Performance of Different Configurations: We also measured the performance of different cores per-VM using 48 cores. As shown in Table 7, the performance actually degrades when the number of cores per-VM increases. The degradation is especially significant due to heavy contention on shadow page table management, with the execution time increasing more than 12X (10.624s vs. 0.860s) when the number of cores per-VM increases from 2 to 8. The evaluation shows that Cerberus does not rely on the scalability of the VMM and can also mitigate performance scalability problems within the VMM when configured properly.

#Cores/VM	Histogram(sec)	Dbench(MB/sec)
2	0.860	2123.6
4	1.130	1805.0
8	10.624	1273.8

Table 7. *Performance of histogram and dbench with different number of cores per-VM*

Performance Comparison with Xen-Linux Shadow Mode: As Cerberus is based on the shadow mode of Xen-Linux, we also give a performance comparison with Xen-Linux for reference. We used Domain0 in shadow mode and direct mode running 32 virtual cores to run histogram and dbench. Due to heavy contention in shadow mode, Xen-Linux experiences extremely bad performance, spending about 246.54s on histogram, and has only 3.4 MB/s throughput for dbench. Yet, for the direct mode, the execution time for histogram is 1.80s and the throughput is 246.54 MB/s for dbench. Hence, when running parallel workloads on multiple cores, it should be better to use direct mode rather than shadow mode. The performance evaluation also shows that Cerberus could not only mitigate the contention within operating systems, but also reduce the contention from multiple cores accessing the shared state owned by a single virtual machine (i.e., shadow page management).

8. Discussion and Future Work

Though Cerberus has demonstrated the applicability of scaling applications with OS clustering, There are still ample optimization and research opportunities remaining. We describe our current limitations as well as possible extensions.

Viability of Our Approach: Our approach is not a panacea to the scalability of applications on multicore, but is only effective in specific scenarios where applications themselves have good parallelism and do not have intensive communication. Specifically, Cerberus might not show performance advantages in the following scenarios. First, applications that clone a number of short-lived, intensively-communicating threads/processes will probably not benefit from our approach, due to the relatively expensive cost of message passing and thread creation. Second, as remote network and remote file introduce overhead in Cerberus, applications with frequent remote resource access might experience degraded performance. Finally, applications with frequent small-size memory mapping operations (e.g., mmap, mremap) will stress the current synchronization mechanism for virtual memory in Cerberus and might have some performance degradation.

Application Cooperation: To retain application transparency, Cerberus relies on some relatively expensive operations (such as inter-VM fork/clone) to support cross-OS execution of an application. In our future work, we would like to investigate ways of adding some appropriate application programming interfaces and libraries to let applications cooperate with Cerberus, thus further reducing the performance overhead. For example, it would be interesting to let user applications explicitly specify which address space range should share the page table, to avoid unnecessary serialization and contention. Moreover, in a fork-intensive application, it would be beneficial for applications to direct Cerberus on which parts need to be checkpointed.

Hardware-assisted Virtualization: Currently, Cerberus is implemented based on hardware platform without hardware-assisted virtualization, thus come with the associated (usually non-trivial) overhead of virtualization. However, hardware-assisted virtualization techniques such as Intel VT-x and AMD SVM with extended page tables or nested page tables are commercially available. Our future work includes incorporating hardware-assisted virtualization to reduce the virtualization overhead, thus further enlarging the performance benefits of Cerberus.

Fault Tolerance: Currently, Cerberus does not provide fault tolerance to applications. While running applications on multiple VMs, it would be desirable if one process fails, processes in other VMs could take over the tasks and proceed as if the failure never happened. However, in Cerberus, if one process of a *SuperProcess* failed in one VM, it is uncertain what would happen to other processes on the other VMs.

9. Conclusions

Scaling operating systems on many-core systems is a critical issue for researchers and developers to fully harness the likely abundant future processing resources. This paper has presented Cerberus, a system that runs a single many-core application on multiple commodity operating systems, yet provides applications with the illusion of running on a single operating system. Cerberus has the potential to mitigate the pressure of applications on the efficiency of operating systems managing resources on many cores. Cerberus is enabled by retrofitting a number of new design techniques back to commodity operating systems to mitigate contention and to support efficient resource sharing. A system call virtualization layer coordinates accesses from process instances in clustered operating systems to ensure state consistency. Experiments with four applications on a 48-core AMD machine and a 16-core Intel machine show that Cerberus outperforms native Linux for a relatively large number of cores, and also scales better than Linux.

10. Acknowledgments

We thank our shepherd Andrew Baumann and the anonymous reviewers for their detailed and insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 61003002, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100, China National 863 program numbered 2008AA01Z138, a research grant from Intel as well as a joint program between China Ministry of Education and Intel numbered MOE-INTEL-09-04, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

References

[Aas 2005] Josh Aas. Understanding the Linux 2.6.8.1 CPU scheduler. `http://joshaas.net/linux/linux_cpu_scheduler.pdf`, Februry 2005.

[Appavoo 2007] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an SMMP OS. *TOCS*, 25(3):6, 2007.

[Barham 2003] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. SOSP*, pages 164–177, 2003.

[Baumann 2009] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schuepbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proc. SOSP*, 2009.

[Boyd-Wickizer 2008] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proc. OSDI*, 2008.

[Boyd-Wickizer 2010] Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Proc. OSDI*, 2010.

[Bugnion 1997] E. Bugnion, S. Devine, and M. Rosenblum. DISCO: running commodity operating systems on scalable multiprocessors. In *Proc. SOSP*, pages 143–156, 1997.

[Chapin 1995] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proc. SOSP*, 1995.

[Chen 2008] X. Chen, T. Garfinkel, E.C. Lewis, P. Subrahmanyam, C.A. Waldspurger, D. Boneh, J. Dwoskin, and D.R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. ASPLOS*, pages 2–13, 2008.

[Engler 1995] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. SOSP*, pages 251–266, 1995.

[Fielding 2002] RT Fielding and G. Kaiser. The Apache HTTP server project. *Internet Computing, IEEE*, 1(4):88–90, 2002.

[Fitzpatrick 2004] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004.

[Franke 2002] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[Gamsa 1999] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. OSDI*, 1999.

[Goldberg 1974] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974.

[Govil 1999] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proc. SOSP*, pages 154–169, 1999.

[Krieger 2006] O. Krieger, M. Auslander, B. Rosenburg, R.W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, et al. K42: building a complete operating system. *ACM SIGOPS Operating Systems Review*, 40 (4):145, 2006.

[Levon 2004] John Levon. *OProfile Manual*. Victoria University of Manchester, 2004. http://oprofile.sourceforge.net/doc/.

[McKenney 2002] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Proceedings of Linux Symposium*, pages 338–367, 2002.

[Mellor-Crummey 1991] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transaction on Computer Systems*, 9(1):21–65, 1991. ISSN 0734-2071.

[Menon 2005] A. Menon, J.R. Santos, Y. Turner, G.J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proc. VEE*, 2005.

[Nightingale 2005] E.B. Nightingale, P.M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. SOSP*, pages 191–205, 2005.

[Nightingale 2009] E.B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proc. SOSP*, 2009.

[PCI-SIG 2010] PCI-SIG. Single-root I/O virtualization specifications. http://www.pcisig.com/specifications/iov/single_root/, 2010.

[Ranger 2007] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. HPCA*, 2007.

[Tridgell 2010] A. Tridgell. Dbench filesystem benchmark. http://samba.org/ftp/tridge/dbench/, 2010.

[Unrau 1995] R.C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*, 9(1): 105–134, 1995.

[Wentzlaff 2008] D. Wentzlaff and A. Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *Operating System Review*, 2008.

[Whitaker 2002] A. Whitaker, M. Shaw, and S.D. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. OSDI*, 2002.

[Zhao 2006] X. Zhao, A. Prakash, B. Noble, and K. Borders. Improving Distributed File System Performance in Virtual Machine Environments. Technical report, CSE-TR-526-06. University of Michigan, 2006.

[Zhong 2001] H. Zhong and J. Nieh. CRAK: Linux checkpoint/restart as a kernel module. *Technical Report CUCS-014-01, Department of Computer Science, Columbia University*, 2001.