VirtuOS: an operating system with kernel virtualization

Ruslan Nikolaev, Godmar Back

SOSP ’13 Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles
Index

- Motivation
- Design & Implementation
- Evaluation
- Conclusion
System critical data of current operating system resides in the kernel’s address space where it cannot be directly accessed by applications.

This isolation falls short of protecting the system from failing kernel components.

Failure of just one kernel component generally causes the entire system to crash.

65~83% of all crashes (in Windows XP)
Motivation

Architectural approaches for increasing the reliability of kernel software and reducing the impact of faults often rely on decomposition.
VirtuOS’s goal

to explore opportunities for improved isolation of kernel components in virtualized containers without significant compromises in performance.
Design & Implementation

- Microkernel
  - Microkernel-based system design moves device drivers and other system critical code from the kernel into separate user space processes.
  - But they require careful engineering to achieve good IPC performance.
  - Extensive emulation interface layers must be implemented.

- 1986 Mach
- 1995 L4
- 2000 Sawmill
Design & Implementation

- **Virtual Machines**
  - also use hardware isolation to create strongly isolated domains in which to separate software components.
  - to safely share a machine’s resources while maintaining application and kernel compatibility with existing systems.
Design & Implementation

Microkernel

Virtual Machines

Dissimilar Motivations
But the same larger design space
Design & Implementation
Design & Implementation

Failure Model

► Primary Domain
  ► A single, primary domain is dedicated to core system tasks such as process management, scheduling, user memory management, and IPC.

► Service Domain
  ► do not run any user processes other than for bootstrapping and to perform any necessary system management tasks related to a domain’s function.
  ► is to handle requests coming from the user processes managed by the primary domain.
Design & Implementation

System Call Design
- Remote system call
  - system calls destined for a service domain
- Local system call
  - directly handled by the primary domain

A modified C library contains all necessary infrastructure to transparently demultiplex local and remote system calls and forward remote calls to service domains.
Design & Implementation

System Call Design

- **Traditional system call implementations** has
  - Mode Switch: transition the processor from a less privileged user mode to kernel mode
  - Cache pollution: Cache and TLB pollution caused by the different working sets of user and kernel code.

So this implementation was discarded, the Exception-less system calls is selected.
Design & Implementation

System Call Design

- Exceptionless System Call (FlexSC: Flexible System Call, 2008)
  - A user-level library places system call requests into a buffer that is shared with kernel worker threads that execute the system call on the task’s behalf, without a mode switch.

- Effective exceptionless system call handling assumes that kernel worker threads run on different cores from the user threads they serve.
Design & Implementation

System Call Design

(a) Traditional, sync. system call

(b) Exception-less system call
Design & Implementation

System Call Design

- impossible direct access in VirtuOS
  - The kernel worker threads reside in a different virtual machine.
  - Our implementation addresses these differences
- which requires the primary domain to communicate essential information about running processes to the service domains
Design & Implementation

System Call Design

Each process creates two such shared areas for each service domain:

1. One area to hold the request queue for outstanding system call requests.
2. An area used as a temporary buffer for system calls that transfer user data.
System Call Design

Copy Strategy

- the user process copies data into or out of a Temporary buffer of memory shared with the service domain.
- A shared memory region is mapped in a continuous virtual address space region in the user program and in the service domain.
Design & Implementation

- Thread Management
  - User-level Thread Scheduling
    - When a system call request is placed into a service domain’s request queue, the issuing user-level thread includes a pointer to its thread control block in the system call entry.

Figure 2: Sharing ready and request queues
Design & Implementation

- Thread Management
  - User-level Thread Scheduling

1. issue a system call
Design & Implementation

- Thread Management
  - User-level Thread Scheduling

Case 1. other threads are ready to execute

If other user-level threads are ready to execute, the current user-level thread blocks and performs a low-overhead context switch to the next ready user-level thread.
Thread Management

User-level Thread Scheduling

Case 2. there are no ready threads

If there are no ready user-level threads after a system call request is issued, a user-level thread spins for a fixed amount of time.

Otherwise, it blocks the underlying kernel thread via a local system call.
Design & Implementation

- Thread Management
  - Worker Thread Scheduling
    - Create worker threads on demand as system call requests are issued, buy always maintain on spare worker thread per process.

---

**Service Domain**

Worker Thread Worker Thread Worker Thread Worker Thread

Request queue Request queue
Thread Management

Worker Thread Scheduling

- Once created, a worker thread remains dedicated to a particular process.
- This fixed assignment allows us to set up the thread’s process specific data structures only once.
Design & Implementation

- Thread Management
  - Worker Thread Scheduling
    - All worker threads cooperate in checking for new requests using the following strategy.
    - When a worker thread has completed servicing system call, it checks the request queues of all other processes for incoming requests and wakes up worker threads for any processes whose request queue has pending requests.
    - Finally, it checks its own process’s request queue and handles any pending requests.
    - If no request is pending in any queue, the worker thread will continue to check those queues for a fixed spinning threshold. If the threshold is exceeded, the worker thread will block.
Design & Implementation

Service Domain

Worker Thread  Worker Thread  Worker Thread  Worker Thread
Request queue  Request queue

complete!
Design & Implementation

Service Domain

Worker Thread  Worker Thread  Worker Thread  Worker Thread
Request queue  Request queue

other process’s request queue check if it has pending requests
Design & Implementation

Service Domain

Worker Thread
Worker Thread
Request queue

Worker Thread
Worker Thread
Request queue

wake up
Design & Implementation

Service Domain

check its own request queue if pending request exists
- If no request in queue, thread is spinning
- If the threshold is exceeded, the worker thread will block.
Evaluation

- System Configuration

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

- **Overhead**
  - **System Call Dispatch & Spinning**
    - For a single `fcntl(2)` call, at least 45 times iteration is needed for the system call to complete without blocking.
    - Without blocking, the achieved throughput was 0.7x that of the native case.
    - If the process is blocked, the throughput slowdowns up to 14x.

- They found to need a much larger spinning threshold (1000 iterations) to achieve the best performance.
- They use the same value for all benchmarks: 1000 iterations.
Evaluation

- Overhead
  - Copying - writes 16MB in chunks of 32, 64, up to 2MB to a file created in a tmpfs filesystem

![Graph 1](image1)
![Graph 2](image2)

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

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

- Overhead
  - Process Coordination
    - A microbenchmark that forks N concurrent processes, then waits for all of them.

*Figure 5: Process Creation Overhead*
Evaluation

Performance

Multithreaded programs

- OLTP/SysBench
  - to evaluate the performance of VirtuOS’s network domains.
  - a MySQL server receives and responds to 10000 requests, each comprising of 10 selection queries with output ordering.

FileIO/SysBench

- to evaluate the performance of VirtuOS’s storage domain
- generates 128 files with 1GB of total data and perform random reads with a block size of 16KB.
Evaluation

- Performance
  - Multithreaded programs - OLTP/SysBench

![Graph showing OLTP/SysBench MySQL throughput](Figure 7: OLTP/SysBench MySQL throughput)
Evaluation

- Performance
  - Multithreaded programs - FileIO/SysBench with and without disk access

Figure 9: FileIO/SysBench throughput without disk accesses
Figure 10: FileIO/SysBench throughput with disk accesses
Evaluation

- Failure Recovery
  - VirtuOS supports failure recovery for any faults occurring in service domains, including memory access violations, interrupt handling routine failure, deadlocks.

![Figure 13: Failure recovery scenario]
Conclusion

- VirtuOS is a fault-resilient operating system design which provides isolation for kernel components by running them in virtualized service domains.

- VirtuOS is the first system to use virtual machines for system call dispatch and to apply exceptionless communication across virtual machines.
Corey : An Operating System for Manycores

Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein MingWu Yuehua Dai Yang Zhang, Zheng Zhang

Silas Boyd-Wickizer et al., 2008 Proceedings of the 8th USENIX conference on operating systems design and implementation
Previous OS

Processor Organizations

- Single Instruction, Single Data Stream (SISD)
- Single Instruction, Multiple Data Stream (SIMD)
- Multiple Instruction, Single Data Stream (MISD)
- Multiple Instruction, Multiple Data Stream (MIMD)

  Uniprocessor

  - Vector Processor
  - Array Processor

  - Shared Memory (tightly coupled)
  - Distributed Memory (loosely coupled)

  Clusters

  - Symmetric Multiprocessor (SMP)
  - Nonuniform Memory Access (NUMA)
Processor trends
Performance Scalability of a Multi-core Web Server

- Length of code path may increase with number of cores
- Examples
  - Waiting longer for spin locks
  - Traversing larger data structures
- Increases instructions per cycle (IPC)
- In fact, IPC is decreasing
Overview

- Paper presents a technique allowing multicore architectures to overcome memory access bottlenecks.
- Key idea is that applications should control sharing of memory and kernel resources.
  - Make them private by default.
  - Let each application specify which resources it want to share.
Introduction

- Throughput of microbenchmark actually decreases with number of cores
- Problem caused by cache coherence protocol
  - Each iteration results in a cache miss
  - Resolving the miss requires access to a shared data structure protected by spin locks
  - Increasing the number of threads attempting to update the table introduces queuing delays

![Throughput of the file descriptor dup and close microbenchmark on Linux](image)
Why throughput drops?

- Load fd_table data to C0 from L1 in 3 cycles.
- Load fd table data to C1 from C2’s L1 in 121 cycles

- AMD 16-core system
  - Sixteen cores on four chips
  - Each core has a 64-KB L1
  - 512-KB L2 cache
  - Each chip has a 2-MB shared L3 cache
The bottleneck is shared OS data structures

- Contention on shared data structures is costly:
  - serialization
  - moving data between caches
- Why does the OS need shared data structures?
  - OS semantics requires it
  - Simplifies resource management
- An example
  - Shared FD table is a bottleneck
    - A lock serializes updates to fd_table
Application spend time in kernel

- Even applications implemented with multicore MapReduce spend time in kernel
  - 30% of time spent in OS on 16 cores

- Fraction of time in OS increases with the number of cores
  - OS becomes a bottleneck
Performance issues

- **Linux spin locks**
  - Repeatedly access a shared lock variable

- **MCS locks** (Mellor-Crummey and Scott, 1991)
  - Process requesting the lock inserts itself in a possibly empty queue
  - Waiting processes do not interfere with each other

- **Spinlocks are better at low contention rates**
  - Require three instructions to acquire and release a lock
  - MCS locks require fifteen instructions

- **MCS locks are much better at higher contention rates**
  - Less synchronization overhead
Performance issues

- Time required to acquire and release a lock on a 16-core AMD machine when varying number of cores contend for the lock.
- The two lines show Linux kernel spin locks and MCS locks (on Corey).
- A spin lock with one core takes about 11 nanoseconds; an MCS lock about 26 nanoseconds.
08’s practices for scaling OS

- Avoiding shared data structures altogether
- Redesign kernel subsystem
  - Fine-grain locking
  - Waith-free primitives
  - RCU
Corey’s solution

- Applications don’t always need to share all the data structures that existing interfaces share
- We propose three interface changes
  - shares, address ranges, kernel cores
- Implemented in Corey OS
  - Partially implemented in Linux
New OS interfaces

- **Shares**
  - control the kernel data used to resolve application references.

- **Address ranges**
  - control page tables and the kernel data used to manage them.

- **Kernel cores**
  - allow applications to dedicate cores to running particular kernel functions.

- **Improve scalability of some applications by avoiding kernel bottlenecks**
Address ranges - The problem

- Two options for multiprocessor
  - Shared address space
  - Private address space
Address ranges - The problem

- Two options for multiprocessor

Shared address spaces

Private address spaces
Address ranges - The problem

- Two options for multiprocessor

Shared address spaces

- `mm_struct`: memory management struct
- `pgtable`: page table

Private address spaces
Address ranges - The problem

- Neither option accurately represents how the application is using kernel data structures:
  - shared address spaces - the mm_struct is global
    - contention
    - unnecessary for private memory
  - private address spaces - the mm_struct is private
    - extra soft page faults, because no PTE sharing
Corey’s solution : address ranges

- **Address ranges** provide benefits of both shared and private address spaces
  - avoid contention for private memory
  - share PTEs for shared memory

※ ar_struct : address range struct
Kernel Cores - The problem

- Application code invokes a system call, kernel code is executed on the same core
  - Uses kernel data structures
  - Acquires locks
- When the kernel data structures are again accessed from a different core, result in cache invalidation
Corey’s solution: Kernel Cores

- Application can dedicate **cores to kernel functions** and data
Corey’s solution: Shares

- Allow applications to dynamically create lookup tables and decide how these tables are shared
- Applications specify when they need sharing, for example:
  - shared FDs allocated in shared table
  - private FDs allocated in private table
- Corey kernel uses shares for all lookup tables
Shares - Linux FD share example

- Cores manipulate FDs without contending for kernel data structures

```plaintext
1. fd2 = open("goo");
3. fdtable1 = share_alloc();
4. fd0 = open("foo", share1);
5. write(fd0, buf, 128, share1);

2. write(fd2, buf, 128);
```
System services

- cfork
  - cfork(core_id) is an extension of UNIX fork() that creates a new process (pcore) on core core_id
  - Application can specify multiple levels of sharing between parent and child
    - Default is copy-on-write

- Network
  - Applications can decide to run
    - Multiple network stacks
    - A single shared network stack

- Buffer cache
  - Shared buffer like regular UNIX buffer cache
  - Three modifications
    - A lock-free tree allows multiple cores to locate cached blocks w/o contention
    - A write scheme tries to minimize contention
    - A scalable read/write lock
Applications - MapReduce applications
Applications - Inverted index with MapReduce
Applications in Address Range-Inverted index with MapReduce

During Reduce PTEs are shared

Avoids contention when growing the address space during Map
Implementation

- Corey runs on AMD Opteron and Intel Xeon processors
- Implementation simplified by using 64-bit virtual address space
- Implementation
  - Low-level Corey objects
    - 11000 lines C and 150 lines assembly
    - Architecture specific functions and device drivers
  - High-level UNIX like environment
    - 11000 lines of C/C++
    - Buffer cache, cfork, and TCP/IP stack interface as well as the Corey-specific glue for the uClibc C standard library, 1wlp, and the Streamflow dynamic memory allocator
Evaluation - Experimental setup

- AMD 16-core system
- 64GB memory
- AMD hardware event counter
  - Number of cache misses
  - Average of latency of cache misses
- OS: Debian
- Kernel patched with perfctr 2.6.35 to allow application access to hardware even counters
Evaluation - Address ranges

- To be investigated:
  - Contention costs of manipulating mappings for private memory
  - Soft page-fault costs for memory that is used on multiple cores

- Expectation:
  - Corey has low costs for both situations
  - Other system have low cost for only one type of sharing, but not both
Evaluation - Address ranges

- Two micro benchmark:
  - Memclone: has each allocate its own 100MB array and modify each page of the array
  - Mempass: Allocates a single 100MB array on one of the clones, touches each buffer page and passes it to the next core which repeats the process

<table>
<thead>
<tr>
<th>Cycles per page</th>
<th>Cores</th>
<th>L3 cache misses</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux single</td>
<td>0</td>
<td>Linux separate</td>
<td>0</td>
</tr>
<tr>
<td>Linux separate</td>
<td>0</td>
<td>Corey address</td>
<td>0</td>
</tr>
<tr>
<td>ranges</td>
<td>0</td>
<td>ranges</td>
<td>0</td>
</tr>
</tbody>
</table>
Evaluation - Kernel Cores

- Two server configurations to be compared:
  - Dedicated: It uses a kernel core for all network processing
  - Polling: It uses a kernel core only to poll for packet notifications and transmit completions
Evaluation - Shares

- Two microbenchmark:
  - Add a per-core segment to a global share with `share_addobj` and Remove segment from the share with `share_delobj`
  - Same but per core segment is added to a local share

![Throughput Graph](image1)

![L3 cache misses Graph](image2)
Evaluation - Applications - MapReduce

- Linux
  - Cores share a single address space
- Corey
  - Each core maps the memory segments holding intermediate results using per-core shared address ranges

Corey and Linux performance

Corey improvement over Linux
Discussion and Future Work

- Corey lacks many features of commodify operation systems, such as Linux, which influences experimental results both positively and negatively.
Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels

- If a process creates a thread with the CLONE_FILES flag, the new thread has its own space of file descriptors; this can reduce sharing and the associated contention

- OpLog Design
  - OpLog optimizations: Batching updates, Absorbing updates, Allocating logs
Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling

- Multicore Research
  - Corey proposes three new abstractions (address ranges, shares and kernel cores), to scale a MapReduce application (i.e., Word Revert Index) running
The Multikernel: A New OS Architecture for Scalable Multicore Systems

- Shared address spaces
  - the former is typically more efficient, however the latter may reduce cross-processor TLB invalidations
Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores

- Related work
  - work has been done to investigate operating systems for multicore processors. One example is Corey which focuses on allowing applications to direct how shared memory data is shared between cores
Efficient System-Enforced Deterministic Parallelism

- Race-free system namespaces
  - This principle ensures that naming a resource reveals no shared state information other than what the application itself provided.
  - Since implicitly shared namespaces often cause multiprocessor contention, designing system APIs to avoid this implicit sharing may be synergistic with recent multicore scalability work.
Q & A