Fast and Correct Performance Recovery of Operating Systems Using a Virtual Machine Monitor

Kenichi Kourai
Kyushu Institute of Technology
kourai@ci.kyutech.ac.jp

Abstract

Rebooting an operating system is a final but effective recovery technique. However, the system performance largely degrades just after the reboot due to the page cache being lost in the main memory. For fast performance recovery, we propose a new reboot mechanism called the *warm-cache reboot*. The warm-cache reboot preserves the page cache during the reboot and enables an operating system to restore it after the reboot, with the help of a virtual machine monitor (VMM). To perform correct recovery, the VMM guarantees that the reused page cache is consistent with the corresponding files on disks. We have implemented the warm-cache reboot mechanism in the Xen VMM and the Linux operating system. Our experimental results showed that the warm-cache reboot decreased performance degradation just after the reboot. In addition, we confirmed that the file cache corrupted by faults was not reused. The overheads for maintaining cache consistency were not usually large.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management

General Terms Design, Performance, Reliability

Keywords Reboot, Page Cache, Performance Degradation, Cache Consistency

1. Introduction

Operating systems are frequently rebooted to recover the whole system. When an operating system crashes due to Mandelbugs [8], it can usually recover from the crash after being rebooted. Since the causes of Mandelbugs are very complex, the rebooted operating system rarely crashes again. The reboot is also used as a simple method for software rejuvenation [7, 10]. Software rejuvenation is a proactive technique to counteract software aging, which is the phenomenon that the state of running software degrades with time. Even if an operating system slows down due to aging-related bugs [8] such as memory leaks, the rebooted operating system can easily restore its normal state.

However, the system performance largely degrades just after the reboot of an operating system. The primary cause is to lose the page cache, namely, disk cache. An operating system stores file contents in the main memory as the page cache to speed up file accesses. When an operating system is rebooted, the contents of the main memory are erased and the page cache is lost. Without the page cache, an operating system has to read files from slow disks. Worse, if the operating system runs in a virtual machine (VM), such cache misses may greatly affect the whole system performance because disks are shared between VMs. The conflicts of disk accesses degrade the performance of not only the rebooted VM but also the other VMs.

Thus, recovery by the reboot is not complete until the system performance is recovered. When an operating system is booted and all the applications on top of it are started, the system can start providing the same services as before the reboot. To make this reboot procedure faster, many techniques have been proposed [2, 12, 20]. However, the system does not restore the same performance at that time. For example, server processes can accept new connections, but they may not return quick responses due to performance degradation caused by frequent cache misses. Such performance degradation lasts until the page cache is re-filled. The size of the page cache tends to increase as the main memory becomes larger.

For fast performance recovery, we propose a new reboot mechanism with the help of a virtual machine monitor (VMM), which is called the *warm-cache reboot*. The warm-cache reboot preserves the page cache in the main memory during the reboot and enables an operating system to restore it after the reboot. This mechanism prevents performance degradation caused by frequent cache misses. To maintain the consistency of the page cache after the reboot, our VMM makes sure that the contents of the page cache are the same as those of corresponding files on disks. A software layer like a VMM is necessary to preserve the page cache in an operating system through its reboot and to reuse only consistent pages even after the crashes of an operating system.

We have developed *CacheMind* on the basis of Xen [3] and implemented the warm-cache reboot mechanism in the VMM and the Linux operating system running on top of it. CacheMind preserves memory allocation to rebooted VMs, manages the page cache through the reboot of operating systems, and maintains the consistency of the page cache. It protects the page cache to prevent operating systems from reusing the corrupted one and reduces the overhead of unprotecting the page cache on file writes by *double caching*. To maintain the cache consistency even for writes to memory-mapped files, we introduced the *unprotect-on-write* bit in page table entries.

From our experimental results, the performance degradation just after the warm-cache reboot was 0 % to 39 % while that just after a normal reboot was 39 % to 90 %. The performance just after the warm-cache reboot was 8.7 times higher at maximum. The overheads for enabling the warm-cache reboot were usually less than 5.5 % for accesses to regular files and less than 13 % for those to memory-mapped files. In worst cases, however, they were 33 % for writes to regular files and 25 % for those to memory-mapped

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'11, March 9–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0501-3/11/03. . . $10.00

files. In addition, our fault-injection test showed that a part of the page cache could be corrupted by faults but it was not reused by the consistency mechanism of the warm-cache reboot.

The rest of this paper is organized as follows. Section 2 describes problems of recovering the system performance after the reboot of an operating system. Section 3 presents the warm-cache reboot. Section 4 explains our implementation based on Xen and Section 5 shows our experimental results. Section 6 examines related work, and Section 7 concludes the paper.

2. Performance Recovery

After the reboot of an operating system, the system performance is largely degraded for a while. The primary cause is various caches being lost in an operating system. Particularly, losing the page cache largely affects the performance. An operating system stores file contents in the main memory as the page cache when it reads them from disks. Since disks are much slower than the main memory, an operating system can speed up file accesses by using the page cache in memory. When an operating system is rebooted, the contents of main memory are erased and the page cache managed by the operating system is lost. Just after the reboot of an operating system, the execution performance of applications running on top of it degrades due to frequent cache misses.

It takes a long time to recover the same performance as before the reboot. To regain the same performance, an operating system has to re-fill the page cache. However, modern operating systems use most of free memory as the page cache to improve performance. The amount of the page cache is almost equivalent to that of free memory, which tends to be larger because the size of memory installable on one machine is increasing due to cheaper memory modules. In addition, widespread 64-bit processors enable an operating system to deal with more than 4 GB of memory. Until the page cache is re-filled, an operating system has to read necessary files from slow disks and cannot recover the performance.

In a VM environment, the performance recovery needs a longer time after an operating system in a VM is rebooted. Recently, server consolidation is performed with VMs for cost efficiency. In such an environment, physical disks are often shared among VMs. Although a different physical disk may be allocated to each VM exclusively, this is usually difficult due to physical constraints or cost. In other words, one VM cannot occupy the whole disk bandwidth. Worse, disk bandwidth allocated to each VM may be limited for fairness. Since this disk sharing degrades the throughput of disk accesses, it takes a longer time to re-fill the page cache by reading files from disks.

Frequent disk accesses affect not only a rebooted VM but also all the other VMs. The conflicts of disk accesses degrade the performance of all the VMs. Just after an operating system in a VM is rebooted, it frequently accesses a physical disk. Increasing disk accesses in one VM affects the performance of the disk access by the other VMs. From the same reason, prefetching does not work well in a VM environment. Prefetching is a common technique for hiding the initial cache misses particularly when the system is booted. Files are read from disks in advance before they are accessed. Prefetching issues too many requests for disk accesses during a short period because it is batch processing, not on-demand. This influences the performance of the other VMs worse.

3. Warm-cache Reboot

To quickly recover the system performance after the reboot of an operating system, we propose a new reboot mechanism called the warm-cache reboot. The warm-cache reboot is achieved by combining an operating system and the underlying VMM. To use the VMM, operating systems run on VMs created by the VMM. A VMM is a useful software layer underlying operating systems to preserve the page cache through the reboot and maintain its consistency.

3.1 Preserving the Page Cache

The basic idea of the warm-cache reboot is to preserve the page cache in memory during the reboot and enable an operating system to restore it after the reboot. We believe that the page cache does not need to be discarded at a reboot. The purpose of a reboot is to initialize the internal state in an operating system or to update its components such as its kernel. Even if the data structures in an operating system are changed by its update, the contents of the page cache are reusable because they are just the copies of file blocks on disks. However, an operating system should not reuse corrupted page cache. Rather, it should read file blocks from disks. The warm-cache reboot discards only corrupted page cache by the consistency mechanism described in Section 3.2.

By reusing the page cache, the warm-cache reboot prevents performance degradation caused by cache misses just after the reboot. In other words, it recovers the system performance as well as the functionality. After the reboot, most of files accessed are expected to exist on the page cache as long as a working set is within the size of the page cache. The workload does not largely change after the reboot because the time needed for the reboot is not very long. Normally, the files accessed during the reboot are not included in the working set just before the reboot. However, the access would just replace a very small part of the page cache in many cases.

While an operating system in a VM is rebooted by using the warm-cache reboot mechanism, the VMM preserves the contents of the memory allocated to the VM. The VMM allocates the same physical memory as before the reboot to the VM. The memory layout is the same as well. Without the VMM, it is not guaranteed that the contents of physical memory are preserved because of a hardware reset. A hardware reset may corrupt a part of memory, depending on hardware [18] and temperature [9]. When the VMM reallocates physical memory, it leaves the contents of the memory as it is. Normally, when the VMM allocates memory pages to VMs, it erases the contents for security. The memory pages may include sensitive information used by another operating system. At the warm-cache reboot, reusing memory pages without erasing the contents is secure. Those pages are necessarily reused for the same operating system.

A rebooted operating system reserves all the pages that have been cached in the page cache before the reboot. We call such memory pages *cache pages*. This reservation is performed at the early stage of booting the kernel, that is, before the kernel starts dynamic memory allocation. This prevents the cache pages from being used for other purposes and corrupted. Since the cache management in an operating system is initialized by the reboot, the VMM manages information to reuse the page cache of an operating system. When an operating system allocates a cache page, it registers the page to the VMM, with the information on the corresponding file block. When an operating system uses that page for other purposes, it unregisters the page. When an operating system is rebooted, it obtains the information on the page cache from the VMM.

3.2 Maintaining Cache Consistency

The warm-cache reboot reuses a cache page only when the page is guaranteed to be consistent. We assume that a cache page is consistent when the contents of the page are the same as those of the corresponding file block on a disk. When a file block is read from a disk to a cache page, the page is consistent. After the page is modified by file writes or destroyed by faults, it becomes

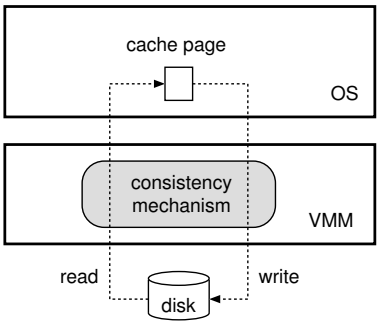

Figure 1. Tracking the consistency of cache pages.

inconsistent. When the cache page is written back to a disk, it becomes consistent again.

The VMM maintains the consistency of each cache page. In a VM environment, device accesses in an operating system running on a VM are performed via the VMM. The VMM reads data on a disk into a cache page passed from an operating system or writes data in a cache page into a disk, as illustrated in Figure 1. When disk reads or writes complete, the VMM makes the cache page reusable because the contents of the cache page are guaranteed to be the same as those of a file block on a disk. We assume that the VMM works as intended and disk reads and writes are performed correctly.

To track the cache consistency, the VMM protects cache pages in a read-only manner. When the VMM reads a file block into a cache page, it protects the page *before* that file has read so that it can detect the modification to the cache page. While a cache page is protected, it is reusable because the cache page is guaranteed to be consistent. This memory protection also prevents a cache page from being corrupted by faults. When an operating system modifies the protected cache page to write data into a file, the VMM changes its protection mode to writable before that write so that an operating system can modify the cache page freely. In this state, the cache page is not reusable because the page is not consistent. When the VMM writes back the contents of the cache page into a disk, it protects the page again before that file write. Thus the cache page becomes reusable again.

The help of the VMM is indispensable to guarantee this cache consistency. Without the VMM, an operating system cannot read a file block on disk into protected cache pages because it cannot write data in protected memory pages. Therefore, the contents of cache pages may be corrupted *during* disk reads because an operating system has to protect cache pages *after* disk reads. If faults make an operating system unstable, the operating system may not protect the cache page correctly. Even if the page is protected correctly, the protection mode may be changed to writable by corrupting the page table. Although an operating system has to manage the reusability of cache pages, such management information may be corrupted accidentally. If that information were wrong, the warm-cache reboot would reuse inconsistent cache pages.

Our assumption for the cache consistency is strong but reasonable. When the page cache is not corrupted, it could be reused even if the contents have not been written back to disk. This enables an operating system to use the latest updates to files after it is rebooted. However, it is difficult to distinguish correct modification from corruption because the correctness of modification depends on semantics. To avoid this semantic problem, we reuse a cache page only after the modification to the page is written back to a disk. Since the modification becomes persistent at that time, the cache page becomes reusable even if its contents are corrupted. In this situation, applications always use the corrupted file even without the page cache. The administrator should recover corrupted files, for example, from the backup.

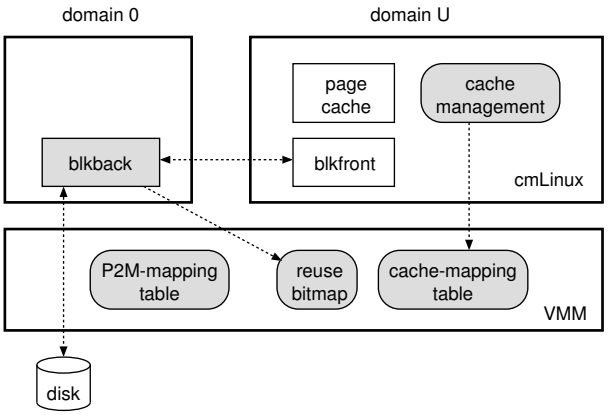

Figure 2. The system architecture of CacheMind.

4. Implementation

We have developed *CacheMind* on the basis of Xen 3.0.0 [3]. Figure 2 shows the system architecture. Xen provides the VMM and VMs running on top of it. A VM is called a *domain* in Xen. Specifically, the privileged VM that manages VMs and handles I/O is called *domain 0* and the other VMs are called *domain Us*. Domain 0 is often considered a part of the VMM. Our modified Linux operating system in domain U is called *cmLinux*. When cmLinux in domain U accesses a virtual disk, its device driver called *blkfront* sends requests to the *blkback driver* in domain 0. The blkback driver accesses a physical disk drive and returns the results to the blkfront driver.

To achieve the warm-cache reboot, our VMM manages a P2M-mapping table, cache-mapping tables, and reuse bitmaps. A *P2M-mapping table* is used for preserving the contents of the memory of domain U even through its reboot. A cache-mapping table and a reuse bitmap are created for each domain U. A *cache-mapping table* is used for restoring the page cache after the reboot. A *reuse bitmap* is used for maintaining the cache consistency.

4.1 Memory Management

In Xen, the VMM distinguishes machine memory and pseudo-physical memory to virtualize memory resources. Machine memory is physical memory installed in the machine and consists of a set of machine page frames. For each machine page frame, a machine frame number (MFN) is consecutively numbered from 0. Pseudo-physical memory is the memory allocated to domains and gives the illusion of contiguous physical memory to domains. For each physical page frame in each domain, a physical frame number (PFN) is consecutively numbered from 0.

A P2M-mapping table is a one-dimensional array that records mapping from PFN to MFN for each domain. In the 64-bit architecture, the table is 2 MB for 1 GB of pseudo-physical memory. A new mapping is created in this table when a new machine page frame is allocated to a domain. When a machine page frame is deallocated, an existing entry is removed. In the current implementation, we do not assume memory overcommitment.

Our VMM preserves the mappings in the P2M-mapping table while a domain is rebooted. In Xen, rebooting a domain destroys the domain and re-creates a new one from scratch. Moreover, if an operating system in a domain crashes, the user has to re-create a

new domain by hand because the original domain is destroyed with the crash. Even when a domain is destroyed, our VMM does not release machine page frames for the domain. If a new domain is re-created with the same domain ID, the VMM allocates the same machine page frames to pseudo-physical page frames in accordance with the P2M-mapping table. The same domain ID is automatically assigned when a domain is rebooted. When a domain crashes, its ID has to be specified by the user to use the same memory mapping. For normally terminated domains, the user can remove all the entries in the table by specifying its ID to release unnecessary memory.

4.2 Cache Management

A cache-mapping table is a hash table whose keys are a tuple of a device number, an i-node number, and a file offset. The value is a PFN assigned to a cache page. When cmLinux accesses files, it first searches its page cache. If no cache page exists in the page cache, cmLinux searches the cache-mapping table in the VMM. If a cache page exists, cmLinux adds a new entry to its page cache so that it can find the cache page quickly the next time. If no cache pages exist in either, cmLinux reads a file block from a disk to a new cache page and adds it to the page cache and the cache-mapping table.

For consistently modifying the cache-mapping table, the VMM provides new hypercalls for cmLinux. After cmLinux reads a file block to a cache page, it adds a new entry to the cache-mapping table by issuing the add_cache hypercall. When it stops using a cache page as the page cache, it removes the corresponding entry from the table by the remove_cache hypercall. These hypercalls perform the sanity check of a request and modify the cache-mapping table consistently. Even if faults make cmLinux unstable, it cannot directly corrupt the table inside the VMM. In addition, such a narrow interface of hypercalls reduces the possibility of incorrectly modifying the cache-mapping table, compared to that of directly corrupting the table without protection by the VMM. Since the VMM updates the table atomically by using the hypercalls, the data structure of the table is preserved correctly. Once the hypercall is issued, it is completed even if cmLinux crashes.

For efficiency, cmLinux can map the cache-mapping table into its kernel address space in a read-only manner. This allows cmLinux to refer to the table directly. If such memory mapping were not performed, cmLinux would have to issue a hypercall even for searching the entries in the table. Such read-only memory mapping eliminates the cost of issuing a hypercall. Since the table is still protected against writes, cmLinux cannot corrupt the table directly. Also, it cannot change its protection mode to writable because the VMM prohibits cmLinux from modifying the page table entries (PTEs) for that memory-mapped table. The VMM can intercept all modifications to the page table by cmLinux.

When cmLinux is rebooted, the VMM first checks its reuse bitmap and removes entries that cannot be reused from the cache-mapping table. Then cmLinux reserves reused cache pages on the basis of the cache-mapping table. The physical address of the table is obtained from the *start_info* page in Xen, which is allocated at the fixed memory location. After cmLinux sets up its page table, it protects reused cache pages on the basis of the table. We assume that no cache pages are corrupted until cmLinux completes the protection. If cmLinux requires new pages but cannot find any free pages due to the reservation of cache pages, it randomly releases the reserved but unused cache pages.

4.3 Reusability Management

A reuse bitmap is a bitmap for maintaining the reusability of cache pages for each domain. Each bit in this bitmap represents whether the corresponding pseudo-physical memory page is reusable as the page cache. A reuse bit is set if the page is used as the page cache and if the contents of the page are guaranteed to be the same as those of a file block on a disk.

4.3.1 Access to Disks

When the blkback driver in domain 0 reads a file block from a disk to a page used as the page cache of cmLinux or writes back a cache page to a disk, it issues the set_reuse_bit hypercall to set a reuse bit for the cache page. This hypercall can be issued only by domain 0. The hypercall sets a reuse bit only if the contents of a cache page are not corrupted during disk I/O. To guarantee this, the page must not be mapped in domain U in a writable manner. If it is, its contents may be corrupted while the blkback driver performs disk I/O. In the current implementation, cmLinux itself protects the cache page just before it sends a request to the blkback driver.

To check that a cache page is not mapped anywhere in a writable manner, our VMM examines the number of writable mapping, which is tracked by the VMM. The VMM assigns a type to each page (such as writable, page table, and so on) and counts that reference. When a page type is writable, the reference count indicates the number of writable mapping. The count is incremented by one whenever a page is mapped in a writable manner in domain U, domain 0, or the VMM. It is decremented by one whenever a writable page is unmapped. The reference count is more than one when there is writable mapping for the page. The count may be more than the actual number of writable mapping because the VMM temporarily increments the value while it manages the page. However, this does not lower the safety.

In addition, the blkback driver checks that a cache page has not been mapped in a writable manner during I/O even temporarily. To do this, the blkback driver issues the set_canary hypercall before starting disk I/O. The hypercall sets a *canary* bit for a specified page if the page is not mapped anywhere in a writable manner. The canary bit shows that the corresponding page has not been mapped in a writable manner. It is cleared once the page is mapped in a writable manner. It is not set again even if the page is unmapped or remapped in a read-only manner. Only domain 0 and the VMM can set the bit. After disk I/O completes, the blkback driver issues the check_canary hypercall to check the canary bit. If the bit is still set, the hypercall sets a reuse bit for a specified page. This guarantees that the page has not been mapped in a writable manner during disk I/O.

4.3.2 Access to Memory-mapped Files

The mmap system call maps cache pages to the address space of a process so that the process can access a file via virtual memory. When a process accesses a memory region where a file is mapped at the first time, a page fault occurs and the kernel maps the corresponding cache page into the region. If no cache page exists, the kernel reads the corresponding file block from a disk. When the mmap system call is issued with the PROT_READ flag, a cache page is mapped in a read-only manner on a page fault caused by the first read access. The VMM does not change the reuse bit for the page because the read-only mapping does not affect the reusability.

On the other hand, when the mmap system call is issued with the PROT_WRITE flag, we have to consider a reuse bit because cache pages can be modified. In the original implementation of Linux, even when a page fault is caused by a read access, the kernel maps the corresponding cache page in a writable manner because the page is permitted to be written. At that time, the VMM clears the reuse bit for the page because its contents may be corrupted. In this implementation, the kernel cannot reuse even cache pages that have not been written. The VMM has no chance to set the reuse bit once a cache page is mapped in a writable manner.

To solve this problem, cmLinux maps a cache page in a read-only manner when a page fault is caused by a read access. At that time, the VMM does not change the reuse bit like the case in which the mmap system call is issued with only the PROT_READ flag. Such a page can be reused safely because the page is guaranteed not to be corrupted. When the page in this state is written, a page fault occurs again and the page is remapped in a writable manner. At the same time, the VMM clears its reuse bit. We call this mechanism *unprotect-on-write*. This is similar to the copy-on-write mechanism, but unprotect-on-write does not copy the contents of the original page to a new page when the page is written. To distinguish unprotect-on-write from copy-on-write, cmLinux sets the unprotect-on-write (UOW) bit in a PTE. For the bit, we used a bit available to operating systems in the x86 architecture.

A cache page modified via a mapped memory region is written back to a disk after the munmap system call is issued to unmap a file. When the blkback driver in domain 0 writes back a cache page, the VMM sets the corresponding reuse bit if possible. The munmap system call does not write back cache pages immediately, but it traverses PTEs and sets a dirty flag to a cache page if a dirty bit is set in the corresponding PTE. A dirty bit in a PTE is set by a write access to the corresponding cache page. Cache pages with a dirty flag are written back periodically or by system calls such as sync.

The other method for writing back modified cache pages is to issue the msync system call explicitly. Like the munmap system call, this system call sets dirty flags to modified cache pages. If the MS_SYNC flag is specified, the system call waits until dirty cache pages are written back. If the MS_ASYNC flag is specified, dirty cache pages are not written back synchronously. To set reuse bits for the cache pages, cmLinux remaps the pages in a read-only manner. If cache pages were mapped in a writable manner, domain 0 could not set their reuse bits. To make the state of these pages unprotect-on-write, cmLinux sets UOW bits for the corresponding PTEs. Since cache pages are protected, a page fault occurs when the first write access is performed to these pages after the msync system call.

Since the mmap system call maps cache pages directly, a process can write data to the region that exceeds the file size. If the file size is not multiples of the page size, the cache page for the end of a file has such a region. To prevent data in such a region from being written back to a disk, the original Linux always fills that extra region with zero when it writes back a modified cache page to a disk. To do this in cmLinux, the kernel has to unprotect the page because the page is not mapped in a writable manner in the kernel address space. To reduce this overhead, cmLinux does not fill the extra region with zero if all the bytes in the region are already zero. This is a normal case because the extra region is zero as long as a process does not modify the region wrongly.

4.3.3 Modification of Page Tables

When the protection mode of a cache page is changed to writable by cmLinux, the VMM clears a reuse bit for the page. Changing a protection mode means modifying a PTE. To virtualize a page table, our VMM supports two mechanisms: the direct page table and the writable page table. If the *direct page table* is used, the VMM can recognize the modification of PTEs by the issues of hypercalls. cmLinux has to issue hypercalls such as mmu_update and update_va_mapping to modify PTEs.

If the *writable page table* is used, the VMM can trap the modification of PTEs. cmLinux can modify PTEs of the writable page table as it directly modifies the table. When it attempts to modify a PTE, a page fault occurs against a page including the PTE because the page table is protected by the VMM. The VMM saves all the PTEs in the page and maps it in a writable manner to the kernel address space. At the same time, the page is disconnected from a page directory entry so that the modification to PTEs in the page is not immediately reflected to the actual page table. The page including modified PTEs is connected to the original page directory entry again after it is revalidated. In the revalidation process, the VMM also clears the corresponding reuse bit if the protection mode of each page is changed to writable.

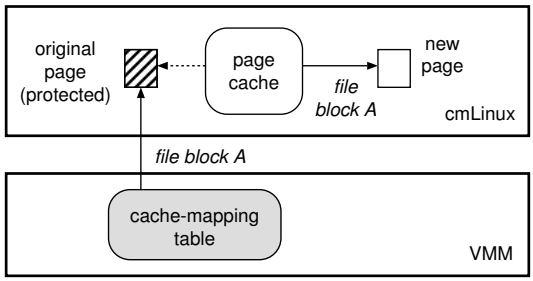

Figure 3. Double caching for reducing write overheads.

4.3.4 Out-of-control Pages

When domain 0 maps a cache page passed from domain U in a writable manner, the VMM clears the reuse bit for the page. For example, let us consider that the netfront driver in domain U erroneously passes a cache page as a buffer to the netback driver in domain 0 for receiving network packets. Like the blkfront and blkback drivers, the netfront and netback drivers communicate with each other to process network packets. If the netback driver overwrites the contents of the cache page with received packets, the corrupted cache page may be reused.

When cmLinux returns a part of memory to the VMM, the VMM clears the reuse bits for the pages. For example, cmLinux returns memory when it receives requests for memory ballooning from the VMM via the balloon driver [23]. It returns several pages and obtains new ones again when it needs contiguous machine memory for DMA. Returning a memory page means that a pseudo-physical page frame does not correspond to any machine page frame. Such returned pages cannot be reused.

4.4 Write Optimization

The write system call is a heavyweight operation in cmLinux because cmLinux has to first unprotect the target cache page if the page is protected. Unlike file reads, the overhead of unprotecting the page is not hidden by a disk access. When cmLinux reads a file block whose cache does not exist in memory, it has to protect a newly allocated cache page, but the overhead of the protection is much smaller than a disk access. However, the write system call without the O_SYNC flag just modifies the page cache in memory and does not require any disk accesses. A disk access is performed later when the dirty cache page is written back. Therefore, the overhead of unprotecting a cache page occupies a large portion of the execution of the write system call.

To eliminate the overhead of unprotecting a cache page during the execution of the write system call, cmLinux temporarily performs *double caching* as in Figure 3. When the write system call is issued, cmLinux allocates a new cache page if the original cache page is protected. It then copies the memory region that will not be modified in the original page to the new one. Finally, it writes data specified by the system call into the new page. At the same time, it replaces the original page in the page cache with the new one. Note that the cache-mapping table is not changed yet. The successive writes to the same file block are performed to that new cache page

without unprotecting it. When the dirty cache page is written back to a disk, cmLinux changes the cache-mapping table, unprotects the original page, and releases it. The overhead of unprotecting the original page is hidden by that disk write. As a side effect, the original cache page is reusable even before the modification to the new page is written back to a disk.

cmLinux performs this double caching only when the written size is bigger than a threshold. The double caching needs memory to be copied from the original cache page to the new one. At worst, cmLinux has to copy 4097 bytes of memory when only one byte is written to a protected cache page. In such a case, the cost of unprotecting a cache page may be less than that of copying a necessary memory region. Since the threshold depends on the MMU performance and memory bandwidth, we can configure the value experimentally.

The double caching does not completely eliminate the overhead in the write system call. First, it still needs extra memory copy. Second, it consumes double the amount of memory used for writes. This memory pressure may cause earlier writeback of dirty pages. To reduce this memory pressure, cmLinux can release the original cache pages before writeback. Third, the double caching cannot be performed when the cache page is already mapped in the address space of a process. To make the double caching consistent, cmLinux also changes memory mapping of the process so that the process refers to a new page. Such an operation, however, is costly.

5. Experiments

We performed experiments to show that the warm-cache reboot is effective. For a server machine, we used a PC with two Dual-Core Opteron processors Model 280, 12 GB of PC3200 DDR SDRAM memory, a 36.7 GB of 15,000 rpm SCSI disk (Ultra 320), and Gigabit Ethernet NICs. We used our VMM and cmLinux based on Linux 2.6.12. For comparison, we used the original Xen 3.0.0 for a normal reboot. We used one physical partition of a disk for a virtual disk of domain U except for experiments in Section 5.4. The Linux ext3 file system was mounted with the ordered mode. We configured several thresholds for writeback so that Linux does not write back dirty cache pages until we issue system calls for writeback, except for experiments in Section 5.5, 5.6, and 5.7. We used the 64-bit execution environment except for experiments in Section 5.7. For a client machine, we used a PC with dual Core 2 Quad processors, 4 GB of memory, and Gigabit Ethernet NICs. The operating system was Linux 2.6.18.

5.1 Effects of Double Caching

We performed an experiment to determine the threshold of the double caching, which we described in Section 4.4. The effects of the double caching depend on the size of the first write to a cache page because cmLinux copies from the original a memory region that will not be written in. In this experiment, we wrote various sizes of data per page and measured the throughputs of the write accesses when we used the double caching and when we unprotected cache pages without using the double caching. Figure 4 shows the throughputs and the performance improvement by using the double caching. The maximum improvement was 37%. The performance improves as the size of written data increases. As an exception, the improvement is not linear when 4096 bytes are written. This is because it is not necessary to issue the lseek system call for moving the file offset to the next 4096-byte block.

However, when the size of written data is less than 1536 bytes, the performance is not improved. In other words, unprotecting a cache page is better than the double caching. Therefore, we used 1536 bytes as the threshold for enabling the double caching in our experiments. If the first write to a cache page is less than 1536 bytes, cmLinux unprotects the cache page and writes data to it directly.

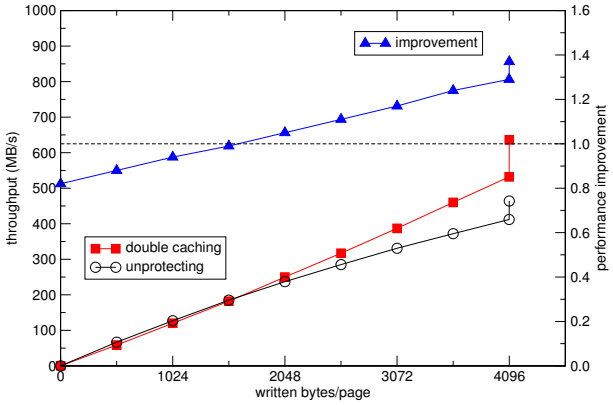

Figure 4. The performance improvement by the double caching for various write sizes.

5.2 Effects of the Warm-cache Reboot

To examine the performance improvement just after the warm-cache reboot, we measured the throughput of file accesses before and after the reboot of an operating system. We accessed one 1-GB file six times and rebooted between the third and fourth accesses. We allocated 4 GB of memory to one domain U and 4 GB to domain 0. In this experiment, all the file blocks were cached in memory. We performed experiments for the warm-cache reboot and the normal reboot.

First, we measured the throughput of file read accesses, changing the block size for file reads. Figure 5 shows the results. For the 4-KB file block, the throughput just after the normal reboot was degraded by 90 % compared with that just before the reboot. The time needed for performance recovery was 8.9 seconds for a 1-GB file. On the other hand, when we used the warm-cache reboot, the throughput just after the reboot was degraded only by 16 %. The throughput is 8.7 times of that just after the normal reboot. The time for performance recovery was only 1.0 second. For the 512-byte block size, the performance degraded by 5.7 %. This improvement was achieved by there being no cache misses in the page cache even when a file was accessed at the first time after the reboot. The remaining performance degradation is caused by misses of other caches in the operating system such as i-node cache.

After the normal reboot, the throughput always becomes worse than that after the warm-cache reboot. The cause is the change of memory allocation to the VM. After the warm-cache reboot, the same memory allocation is preserved by the VMM. However, after the normal reboot, a new VM is created and different ranges of memory are allocated. Since Opteron processors we used adopt non-uniform memory architecture (NUMA), the latency of memory access became large due to this change. Also, the third and fifth accesses are better than the second and sixth, respectively. This is because Linux moves a cache page from the inactive list to the active one.

Second, we measured the throughput of file write accesses. We prepared a 1-GB file in advance and rewrote it repeatedly. Figure 6 shows the results when we changed the block size for file writes. For the block sizes less than 4 KB, the throughput just after the reboot was improved when we used the warm-cache reboot. When the block size is 2 KB, the performance degradation is 92 % in the normal reboot but 37 % in the warm-cache reboot. In this case, the time needed for performance recovery was 31 and 3.8 seconds in

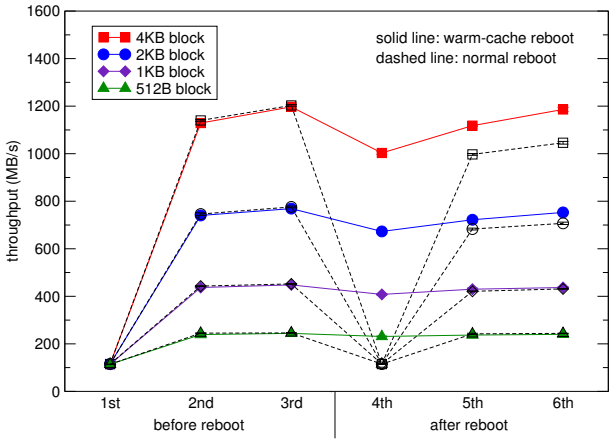

Figure 5. The throughput of file reads through the reboot.

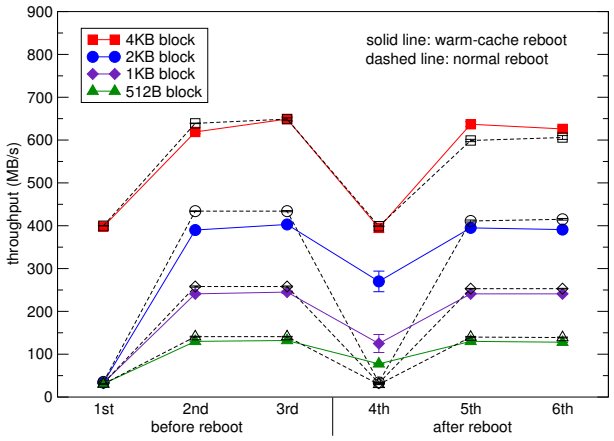

Figure 6. The throughput of file writes through the reboot.

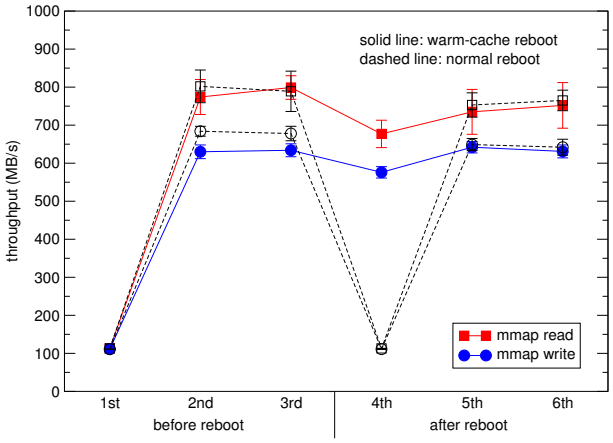

Figure 7. The throughput of accesses to a memory-mapped file through the reboot.

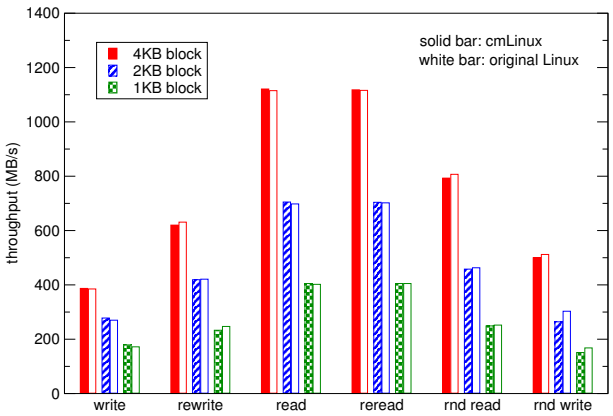

Figure 8. The results of IOzone.

the normal reboot and in the warm-cache reboot, respectively. This performance improvement is caused by a file read before the first write to a new cache page. In the normal reboot, the file read is done from the disk. The warm-cache reboot can reuse the cache page in memory. For the 4-KB block size, on the other hand, both throughputs are almost the same and the degradation is 39 %. Since the block size is the same as the page size, file reads after the reboot are not performed.

Third, we measured the throughput of the accesses to a memory-mapped file. For reads, we mapped a 1-GB file into the memory and read it sequentially. For writes, we mapped a prepared 1-GB file into the memory and rewrote it sequentially. The block size was 4 KB because it did not greatly affect the throughput. Figure 7 shows that the warm-cache reboot can decrease the performance degradation just after the reboot. In case of the normal reboot, the throughputs of reads and writes are degraded by 86 % and 83 % and the recovery times are 9.0 and 9.2 seconds, respectively. For the warm-cache reboot, the throughputs are degraded only by 15 % and 9.1 % and the recovery times are only 1.5 and 1.8 seconds, respectively.

5.3 Overheads

To examine the overheads for enabling the warm-cache reboot, we first executed the IOzone 3.347 benchmark [19] in cmLinux and the original Linux. We specified 512 MB as its file size. We executed sequential write, rewrite, read, and reread and then executed random read and write. We allocated 4 GB to domain U and all the file blocks created temporarily were cached in memory. Figure 8 shows the results when IOzone uses file I/O system calls. The overheads for file reads are negligible. On the other hand, file writes involve overheads. The worst throughput degradation among sequential writes is 5.5% with the 1-KB block size. For random writes, the overheads are 2.3 % to 13 %. Figure 9 shows the results when IOzone uses the mmap system call. The throughput of all accesses is degraded and the overheads are 2.9 % to 8.9 %.

Second, we examined the relationship between the overhead of file writes and the write size per cache page. We changed the write size for each 4-KB file block and measured the throughput of file writes. Figure 10 shows the throughputs in cmLinux and the original Linux and the overhead in cmLinux. For 1-byte write per page, the overheads are 33% and quite large. This is due to unprotecting a cache page for only 1-byte write. However, the overhead decreases as the write size per page increases.

Third, we examined the overhead of unprotect-on-write for a memory-mapped file. We mapped a file to the memory with the PROT_READ and PROT_WRITE flags, read the memory, and then wrote it. In cmLinux, a page fault occurs at the first write after reads to achieve unprotect-on-write. We measured the throughput

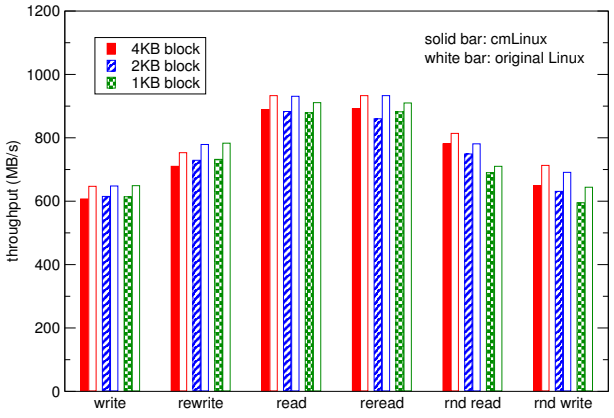

Figure 9. The results of IOzone with mmap.

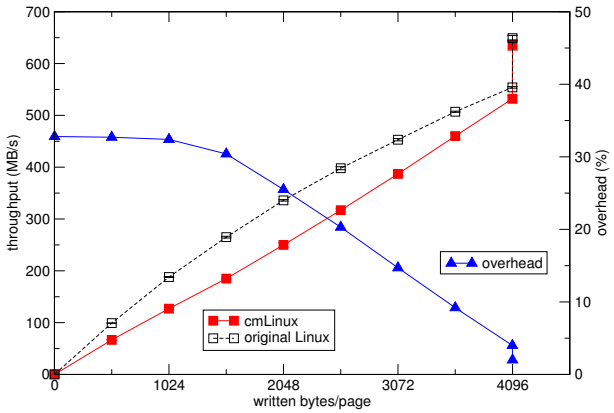

Figure 10. The throughput of partial writes for each cache page.

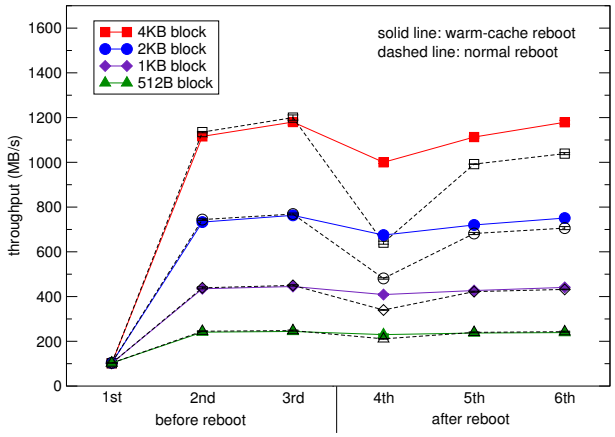

Figure 11. The throughput of file reads from a file-backed virtual disk through the reboot.

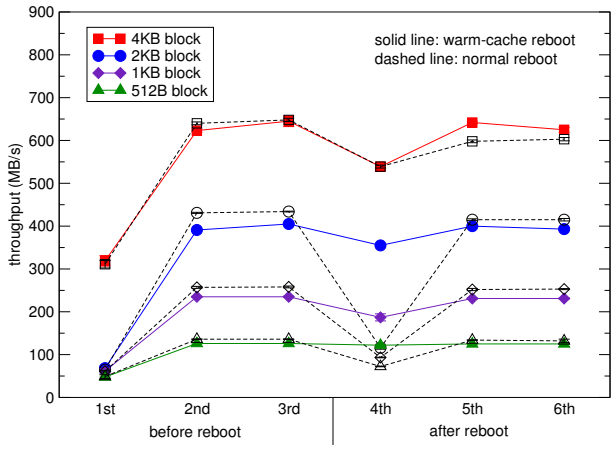

Figure 12. The throughput of file writes from a file-backed virtual disk through the reboot.

of a set of read and write after the file blocks are cached in memory. The throughputs in cmLinux and the original Linux were 344 and 459 MB/s, respectively. Compared with the original Linux, the throughput in cmLinux degraded by 25%.

Finally, we measured the time needed for the writeback in cmLinux and compared it with that in the original Linux. When cmLinux writes back a dirty cache page to a disk, it performs heavyweight operations. It has to protect the page before the writeback. If the double caching is performed, cmLinux also has to modify the cache-mapping table by issuing a hypercall and unprotect the original page. We wrote 1-GB data and wrote back all of the dirty cache pages. To make cmLinux write back dirty cache pages, we issued the fsync system call when we wrote data by using the write system call. For writes to a memory-mapped file, we issued the msync system call for the writeback. The times needed for fsync were 11.0 and 10.9 seconds for cmLinux and the original Linux, respectively. Those for msync were 12.2 and 12.0 seconds, respectively. As a result, the overheads of the writeback are only 0.4% and 1.6% for fsync and msync, respectively. This means that the overheads are hidden by disk accesses.

5.4 Effects of the Page Cache in Domain 0

To examine the effects of the page cache in domain 0, we used a file-backed virtual disk for domain U. In the experiments of the previous sections, we used a partition-based virtual disk. In this configuration, the blkback driver in domain 0 directly reads the physical partition without the interference with any file systems of the operating system in domain 0. On the other hand, when the domain U reads a file block in a file-backed virtual disk, the blkback driver in domain 0 reads the corresponding file block from the image file for a virtual disk. At that time, the operating system in domain 0 also caches the file block. Even if the domain U is rebooted, the file block is still cached in domain 0. When domain U reads a file after the reboot, the blkback driver in domain 0 returns data in the page cache without accessing a physical disk.

Since we allocated 4 GB to domain 0 and domain U for each, all the file blocks were cached in the page cache of both domain 0 and domain U. Figures 11 and 12 show the throughputs of file reads and writes, respectively. When we used the normal reboot, the throughput of file reads degraded by 47 % for the 4-KB file block and that of file writes by 73 % for the 2-KB file block. Compared with when we used a partition-based virtual disk, the performance improved because of the page cache in domain 0. However, the performance degradation was larger than when we used the warm-cache reboot. The blkfront driver had to communicate with the blkback driver in domain 0 and copy file blocks from domain 0 to domain U.

	before reboot		after reboot	
	1st	2nd	3rd	4th
normal reboot	499	168	502	168
warm-cache reboot	509	168	168	167

Table 1. The time for the power test in the DBT-3 benchmark (sec).

Next, we changed the memory size of domain 0 from 4 GB to 1 GB. When we access a 1-GB file in domain U, all the file blocks cannot be cached in the page cache in domain 0. As a result, the throughput degradation of file reads increased from 47 % to 90 %. That of file writes also increased from 73 % to 89 %. These degradation levels are almost the same as when we use a partition-based virtual disk.

By comparing the results in Figure 11 with those in Figure 5, reusing metadata in file systems is found to not improve the performance just after the reboot. When we use a file-backed virtual disk, not only file data but also metadata in the disk image is cached in domain 0. However, the throughput just after the reboot is almost the same as when we use a partition-based virtual disk.

5.5 DBT-3

To examine the performance of more realistic applications, we measured the time needed for the power test with the scale factor of one in the DBT-3 benchmark 1.9 [24] before and after the reboot. DBT-3 tests database performance in a decision support system and it is a simplified implementation of the TPC-H benchmark [22]. Its power test measures the performance of the read access to databases. We used PostgreSQL 8.2.4 as a database. We measured the performance four times and rebooted between the second and third tests. We allocated 11 GB of memory to one domain U and 512 MB to domain 0. All the file blocks were cached in memory in this experiment. We performed this experiment for the warm-cache reboot and a normal reboot.

Table 1 shows the results when the data size was 1 GB. When we used the normal reboot, the performance just after the reboot degraded by 67 % compared with that just before the reboot. On the other hand, when we used the warm-cache reboot, the performance did not degrade at all.

5.6 Web Server

We measured the changes of the throughput of a web server before and after the reboot of an operating system. The Apache web server 2.0.54 [1] served 4000 files of 1 MB, and httperf 0.8 [17] in a client host sent requests to the server one by one. Since we allocated 11 GB of memory to one domain U, all the files served by the web server were cached in memory. We allocated 512 MB of memory to domain 0.

Figure 13 shows the changes of the throughput of a web server when we used the normal reboot and the warm-cache reboot. We executed the reboot command in domain U in 30 seconds. When we used the warm-cache reboot, the throughput was degraded only by 5 % after the reboot. In 60 seconds after the web server restarts its service, the throughput is recovered completely. On the other hand, when we used a normal reboot, the throughput was degraded by 41 % on average. The performance degradation lasts for 90 seconds after the web server restarts its service. During this period, the web server loses the benefit to be gained of about 3300 requests, compared with before the reboot.

5.7 Fault Injection

We injected faults into an operating system in domain U and examined the consistency of reused cache pages. We ported the fault injection tool used in the Nooks [21] project to the Linux 2.6 kernel.

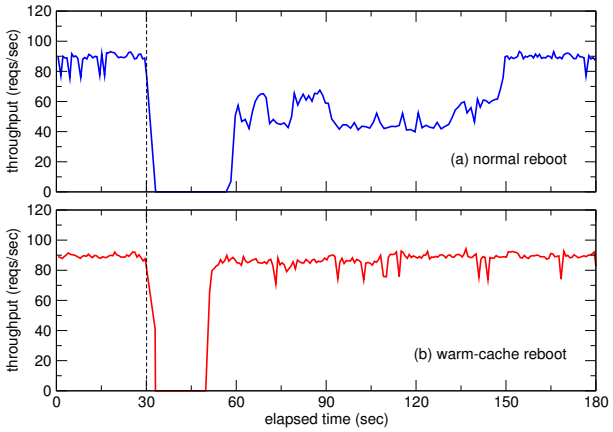

Figure 13. The changes of the throughput of a web server when an operating system is rebooted.

Originally, the tool was developed for the Rio file cache [5] project. Since the tool developed by the Nooks project strongly depends on Intel 32-bit architecture, we used the 32-bit execution environment.

Faults injected by this tool are categorized into three types. The first category is programming errors. *Destination faults* flip a random bit of the destination of an instruction to emulate assignment errors. *Pointer faults* flip a random bit of the address for memory reference to emulate incorrect pointer calculations. *Initialization faults* delete an instruction that initializes a local variable on the kernel stack to emulate the usage of uninitialized local variables. *Interface faults* delete an instruction that reads a function parameter to emulate bad parameters. *Branch faults* delete a branch instruction or a repeat prefix to emulate bugs in control flow. *Loop faults* invert the termination condition for a repeat prefix or a branch instruction. *Panic faults* cause a kernel panic.

The second category is memory management errors. *Allocation faults* return NULL at the kmalloc function to emulate memory exhaustion. *Free faults* release a memory region that is still used. *Memory leak faults* do not release a memory region at the kfree function. *Bcopy faults* overrun the length of memory copy by one byte to four pages.

The third category is memory corruption errors. *Text faults* flip a random bit of a random instruction in the kernel. *Stack faults* flip a random bit in a stack of a random process. *NOP faults* delete a random instruction in the kernel.

We examined the consistency of the page cache after we injected faults into an operating system and rebooted it using the warm-cache reboot. First, we booted an operating system in domain U and waited until the page cache is filled by sending HTTP requests. We allocated 1 GB of memory to domain U and sent requests for 768 MB of files. Then we injected ten faults of the same type into the kernel and waited for 60 seconds. Finally we rebooted the operating system and checked the consistency of the page cache by comparing it with files on a disk. We repeated this fault injection 50 times for each fault type.

Figure 14 shows the ratio at which the page cache was inconsistent when the consistency mechanism was disabled. The VMM did not manage the page cache. Instead, an operating system managed a cache-mapping table and a reuse bitmap without the help of the VMM. For most of fault types, the page cache was inconsistent at a high ratio. This figure also shows the breakdown of the results of this fault injection when the page cache was inconsistent. Although fault injection did not always cause a crash, the page cache became inconsistent.

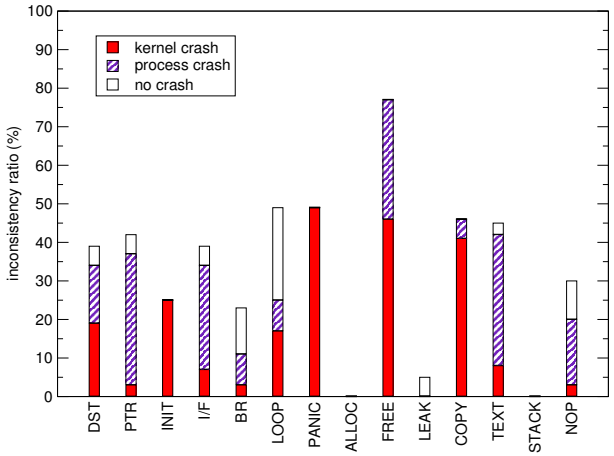

Figure 14. The ratio of cache inconsistency when the consistency mechanism was disabled.

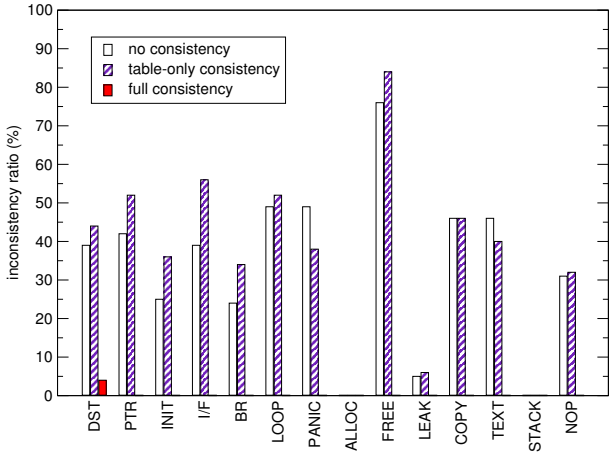

Figure 15. The ratios of cache inconsistency when the consistency mechanism was enabled at various levels.

Figure 15 shows the ratios at which the page cache was inconsistent: (1) when the consistency mechanism was disabled, (2) when the consistency mechanism was enabled only for a cache-mapping table, not for a reuse bitmap, and (3) the consistency mechanism was fully enabled. When only a cache-mapping table was managed by the VMM, the ratio did not decrease. This means that a cache-mapping table is unlikely to be corrupted by faults. When the full consistency mechanism was enabled, the page cache was consistent for all fault types except destination faults (DST). According to our deep inspection, some faults were injected into the ext3 file system in this exceptional case. Then the file system failed to write back cache pages to a disk. This resulted in the inconsistency between the page cache and files on the disk. However, the contents of the page cache were correct while those of files on the disk were incorrect. Therefore, reusing the page cache is correct although the consistency is not maintained.

6. Related work

The Rio file cache [5] enables dirty file cache to survive crashes of an operating system. When an operating system crashes, Rio saves dirty cache pages to a disk and prevents the reboot from losing any modification to files. The biggest difference between Rio and CacheMind is that Rio is designed for reliability while CacheMind is for high performance. When an operating system is rebooted, Rio discards non-dirty file cache because saving it is not necessary for improving reliability. To the contrary, CacheMind reuses non-dirty page cache but discards dirty ones because dirty pages are inconsistent with disks. In addition, because Rio has to read saved file cache from a slow disk, the performance degrades just after the reboot. CacheMind prevents such performance degradation by reusing the page cache preserved in memory.

The other big difference is that Rio relies only on an operating system (and hardware) while CacheMind relies on the VMM. For example, Rio provides two mechanisms to save the file cache to a disk on a crash. One is to perform a warm reboot, which preserves memory contents during the reboot, and save dirty file cache after the reboot [5]. The other is to save the file cache using a BIOS routine before a reboot [18]. The former depends on hardware and is not generally supported in PCs. The latter might fail because Rio cannot always execute the BIOS routine after a crash. In CacheMind, an operating system in a VM can perform a warm reboot, independently of hardware, because the VMM guarantees to preserve memory contents during the reboot of the VM.

Besides, Rio uses memory protection to prevent the file cache from being corrupted by crashes of an operating system. Rio protects the file cache by using functions in an operating system while CacheMind protects it by the VMM. In Rio, if the page table is corrupted by a crash of an operating system, memory protection might be ineffective. In CacheMind, although the page table may be corrupted, the VMM tracks any modification and maintains the reusability of the page cache. Also, Rio cannot atomically modify its registry for cache management because the registry is also managed by an operating system. Therefore, the consistency of the registry is not guaranteed when an operating system crashes. CacheMind manages a cache-mapping table for cache management with the VMM. The modification of the table can be atomically performed by the VMM.

As described in Section 5.4, when a file-backed virtual disk is used in Xen, the page cache in domain 0 helps performance recovery just after the reboot of an operating system in domain U. Even in type-II VMMs such as Linux Kernel-based Virtual Machine (KVM) [13], the page cache in its host operating system is helpful as well. However, the performance still degrades due to the memory copy from the page cache in domain 0 to that in domain U. Another drawback is that domain 0 needs larger memory to cache file blocks for all domain Us. It is not efficient to store the same file blocks both in domain 0 and domain U.

Non-volatile disk cache such as Microsoft hybrid hard drive (HHD) and Intel Turbo Memory are also useful for fast performance recovery. HHD includes non-volatile memory inside a disk drive while Turbo Memory is attached to a motherboard. They enable an operating system to read file blocks from fast non-volatile memory even if the page cache in memory is lost after the reboot. Furthermore, a solid-state drive (SSD) speeds up the whole disk accesses by using flash memory. However, file blocks from non-volatile memory must be copied to the page cache in the main memory. CacheMind does not need any memory copies because it preserves the page cache on the main memory through the reboot.

Geiger [11] maintains the mapping between disk blocks and the page cache only by the VMM without the knowledge of operating systems. Since it infers the mapping using several heuristics, it can miss the detection of cache-page eviction by operating systems. This is critical in our context because such cache pages are reused for other purposes and do not contain valid contents for associated files. Similarly, the time lag between actual page eviction and its detection is also critical. To address this problem, hypervisor

exclusive cache [16] modifies operating systems so as to notify the VMM of page eviction.

Otherworld [6] quickly recovers applications by using the microreboot technique [4]. When the kernel crashes, Otherworld starts another kernel called the crash kernel and restores the state of applications and the operating system, including dirty file cache. The correctness of the recovery mainly depends on the assumption that the probability of the data corruption necessary for recovery is low. CacheMind protects the file cache by using the VMM because the amount of reused non-dirty file cache is large and the probability of cache corruption is high, as shown in Section 5.7.

Recovery Box [2] preserves the state of an operating system and applications on non-volatile memory for fast recovery. It restores the state quickly after rebooting an operating system. The state stored in that memory is protected by checksum. In addition, Recovery Box speeds up a reboot by reusing the kernel image left on memory. This is less effective recently because recent disks are fast enough to read the small file for the kernel image.

RootHammer [14, 15] enables only the VMM to be quickly rebooted by leaving VM images in memory. It uses the fact that VM images can be reused after the reboot of the VMM. Similarly, CacheMind uses the fact that the page cache can be reused after the reboot of an operating system. In addition, CacheMind reuses only consistent page cache with the help of the VMM.

7. Conclusion

In this paper, we proposed a new reboot mechanism, called the *warm-cache reboot*, for fast and correct performance recovery. The warm-cache reboot preserves the page cache on main memory during the reboot and restores it quickly after the reboot. The VMM guarantees that the page cache reused after the reboot is consistent with the corresponding files on disks by maintaining reuse bitmaps. We have implemented the warm-cache reboot mechanism in Xen. According to our experimental results, the performance just after the reboot became 8.7 times higher at most when we used the warm-cache reboot. The overheads for enabling the warm-cache reboot were usually not large. In addition, it was shown that faults did not corrupt the reused page cache. One of our future work will be to reuse other caches in an operating system such as i-node cache to improve the performance just after the reboot.

Acknowledgments

This research was supported in part by JST, CREST.

References

[1] Apache Software Foundation. Apache HTTP Server Project. `http://httpd.apache.org/`.

[2] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *Proceedings of the Summer USENIX Conference*, pages 31–44, 1992.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 164–177, 2003.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, 2004.

[5] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[6] A. Depoutovitch and M. Stumm. Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems*, pages 181–194, 2010.

[7] S. Garg, A. Puliafito, M. Telek, and K. Trivedi. Analysis of Preventive Maintenance in Transactions Based Software Systems. *IEEE Transactions on Computers*, 47(1):96–107, 1998.

[8] M. Grottke and K. Trivedi. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. *IEEE Computer*, 40(2):107–109, 2007.

[9] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the USENIX Security Symposium*, pages 45–60, 2008.

[10] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software Rejuvenation: Analysis, module and Applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 381–391, 1995.

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

[12] H. Kaminaga. Improving Linux Startup Time Using Software Resume (and Other Techniques). In *Proceedings of the Linux Symposium*, pages 25–34, 2006.

[13] A. Kivity, Y. Kamay, and D. Laor. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[14] K. Kourai and S. Chiba. A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines. In *Proceedings of the 37th International Conference on Dependable Systems and Networks*, pages 245–254, 2007.

[15] K. Kourai and S. Chiba. Fast Software Rejuvenation of Virtual Machine Monitors. *IEEE Transactions on Dependable and Secure Computing*, 2010.

[16] P. Lu and K. Shen. Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–15, 2007.

[17] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. *Performance Evaluation Review*, 26(3):31–37, 1998.

[18] W. Ng and P. Chen. The Design and Verification of the Rio File Cache. *IEEE Transactions on Computers*, 50(4):322–337, 2001.

[19] W. Norcott and D. Capps. IOzone Filesystem Benchmark.

[20] A. Pfiffer. Reducing System Reboot Time with kexec. `http://www.osdl.org/`.

[21] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 207–222, 2003.

[22] Transaction Processing Performance Council. TPC Benchmark H Standard Specification Revision 2.9.0. `http://www.tpc.org/`, 2009.

[23] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, 2002.

[24] J. Zhang and M. Wong. Database Test Suite. `http://osdldbt.sourceforge.net/`.