SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **42**:303–330
Published online 30 March 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1069

VM aware journaling: improving journaling file system performance in virtualization environments

Ting-Chang Huang[1] and Da-Wei Chang[2,*,†]

[1]*Department of Computer Science, National Chiao Tung University, No. 1001, University Road, Hsinchu 300, Taiwan*
[2]*Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, Ta-Hsueh Road, Tainan 701, Taiwan*

SUMMARY

Journaling file systems, which are widely used in modern operating systems, guarantee file system consistency and data integrity by logging file system updates to a journal, which is a reserved space on the storage, before the updates are written to the data storage. Such journal writes increase the write traffic to the storage and thus degrade the file system performance, especially in full data journaling, which logs both metadata and data updates. In this paper, a new journaling approach is proposed to eliminate journal writes in server virtualization environments, which are gaining in popularity in server platforms. Based on reliable hardware subsystems and virtual machine monitor (VMM), the proposed approach eliminates journal writes by retaining journal data (i.e. logged file system updates) in the memory of each virtual machine and ensuring the integrity of these journal data through cooperation between the journaling file systems and the VMM. We implement the proposed approach in Linux ext3 in the Xen virtualization environment. According to the performance results, a performance improvement of up to 50.9% is achieved over the full data journaling approach of ext3 due to journal write elimination. In metadata-write dominated workloads, this approach could even outperform the metadata journaling approaches of ext3, which do not guarantee data integrity. These results demonstrate that, on virtual servers with reliable VMM and hardware subsystems, the proposed approach is an effective alternative to traditional journaling approaches. Copyright © 2011 John Wiley & Sons, Ltd.

Received 5 May 2010; Revised 26 January 2011; Accepted 28 January 2011

KEY WORDS: journaling file systems; virtual machines; file system consistency

1. INTRODUCTION

File system consistency and data integrity are the most critical issues for file system design. When a system crashes, partially flushed file operations may lead to file system inconsistency and file data corruption. Journaling file systems address this problem based on the concept of Write-Ahead Logging (WAL) [1]. Specifically, a journaling file system logs the file system updates to a *journal area*, or simply a *journal*, before they are written to the data storage. As a consequence, the file system can be efficiently brought back to a consistent state by replaying the *journal data* (i.e. data in the journal area). Because of the efficient recovery, journaling file systems, such as ext3 [2], ReiserFS, XFS, JFS and NTFS, are widely used in modern operating systems.

Generally, two kinds of journaling approaches are supported in commodity journaling file systems: metadata journaling and full data journaling. Unlike metadata journaling that ensures

*Correspondence to: Da-Wei Chang, Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, Ta-Hsueh Road, Tainan 701, Taiwan.
†E-mail: dwchang@mail.ncku.edu.tw

only file system consistency, full data journaling guarantees file system consistency as well as data integrity by logging both data and metadata updates. Although data integrity has become a critical issue in recent file systems [3, 4], the high overhead of full data journaling restricts it from being a commonly used approach. Specifically, journaling all the data and metadata updates doubles the write traffic to the storage and thus degrades the file system performance under most workloads [5, 6].

In this paper, we propose a new journaling approach called Virtual Machine Aware journaling (VMA journaling) that supports full data journaling while eliminating all the journal writes (i.e. writes to the on-storage journal area) in virtual server environments, which are gaining in popularity in server platforms. In the recent years, virtual machines have been broadly used in server consolidation, disaster recovery, software testing, security and storage management. According to International Data Corporation (IDC), in 2008, there were more shipments of servers based on virtual machines than those based on physical machines, and the ratio will reach 1.5 in 2013 [7]. Moreover, previous studies have suggested running operating systems and applications on virtual machines to enhance security, mobility and reliability [8, 9]. In a virtual server environment with reliable VMM and hardware subsystems, VMA journaling provides superior file system performance with similar level of data integrity and consistency when compared with traditional full data journaling. This is achieved by allowing cooperation between the journaling file systems and the Virtual Machine Monitor (VMM) [10, 11], which is a thin software layer running underneath the virtual machines and emulating the hardware interface to the guest operating systems (i.e. the operating systems running in the virtual machines). Unlike traditional journaling approaches, which write journal data to the on-storage journal area, VMA journaling retains journal data in the memory of the virtual machine and stores the information for locating these data (called the *journal information*) in the VMM. As a consequence, journal writes are eliminated. In addition, the memory overhead is not significant since only the metadata for locating the journal data are maintained in the VMM memory. Moreover, page protection is used to prevent wild writes from corrupting the journal data.

VMA journaling targets at virtual server platforms with reliable VMM and hardware subsystems. Specifically, VMA journaling is based on the following two assumptions. First, hardware errors do not lead to loss of critical data. This assumption is similar to the one made in traditional journaling file systems that no faults in the storage subsystem lead to data loss [12]. Hardware subsystems are usually reliable in server platforms due to redundancy-based techniques, such as redundant power, memory mirroring, RAID, or error detection/handling mechanisms in modern architectures. For example, power outages can easily be addressed by battery backup components (e.g. UPS). When power outages occur, VMM can be notified to start the regular shutdown procedure, including flushing critical information back to the storage. As another example, many modern architectures, especially those that target server platforms, such as the Intel Xeon processor 7500 series, support RAS (reliability, availability and serviceability) features [13]. In these architectures, malfunctions of hardware components can be detected or handled by the hardware. CPU malfunction can be detected by the machine check exception [14] and handled in the VMM by CPU offline. Memory errors can also be handled by error correction techniques or memory mirroring [15, 16]. Furthermore, uncorrectable memory errors are usually preceded by correctable errors [17]. Thus, VMM can proactively move critical information out of a memory area when correctable memory errors have been detected in that area, further reducing the probability of losing critical information due to uncorrectable memory errors. Support of RAS features has been included in many VMMs, such as VMware ESX server and Xen [18].

Second, sudden VMM crashes do not occur. This assumption is supported by the low complexity of a VMM. Many VMMs, such as OKL4, NOVA, Hyper-V, have similar architecture with microkernel or were designed based on the concept of microkernel [19–23], which exposes a narrow interface and has a small code size. Complicated or error-prone operating system components, such as network protocol stacks, file systems and drivers, are moved out of the kernel to achieve higher reliability. For example, moving device drivers, which are known to be the primary source of bugs that lead to system failure [24], out of the VMM greatly helps to reduce the risk of VMM

DOI: 10.1002/spe

crashes. As indicated by a previous study [19], the similarity between microkernels and VMMs are growing. Research efforts are made to reduce the code sizes of VMMs [23, 25] while microkernels are increasingly used as VMMs [19, 26]. This previous study also shows that the requirements of both a microkernel and a VMM can be met with a single implementation. In addition, hardware vendors such as Intel and AMD also contribute to the reduction of VMM complexity by providing hardware-supported virtualization technologies [27, 28]. The low complexity of VMMs allows them to be verified or secured easily. Recently, a micro-kernel seL4 [29], which can be used directly as a VMM [30], has been proved functionally correct by formal verification, meaning that the kernel never crashes, falls into unknown states or performs unsafe operations. Therefore, bug-free VMMs are currently technically feasible [31]. Since the reliability and security of VMMs are increasingly being addressed [23, 25, 32], we expect more VMMs to be verified in the future.

Although VMMs can be made reliable, the high complexity of full-blown operating systems prevents them from being verified as correct [24]. For example, Linux 2.6.29 has about 7 million SLOC (source lines of C), making the verification nearly impossible currently or in the foreseeable future. Therefore, virtual machine crashes still occur. For example, buggy device drivers, malicious viruses or network attacks can crash a virtual machine. VMA journaling aims to ensure file system consistency and data integrity in the case of virtual machine crashes. The journal data remain intact during a virtual machine crash since we retain the physical memory of the crashed virtual machine temporarily (before a complete flush of the journal data) and use page protection to prevent wild writes to the journal data. Since the integrity of the journal data/information is maintained, file system recovery can be achieved after virtual machine crashes.

We have implemented VMA journaling on the ext3 file system as a loadable module in Linux and modified Xen, an open-source VMM, to support VMA journaling. The current implementation adds less than 600 lines of C code into the Xen VMM and hence does not cause a noticeable increase in the complexity of the VMM. Moreover, the proposed journaling interface allows other journaling file systems to enjoy the benefits of VMA journaling without much effort. According to the performance results, VMA journaling achieves a performance improvement of up to 50.9% when compared with the full data journaling approach of ext3. In workloads dominated by metadata writes, VMA journaling even shows performance superior to the metadata journaling approaches of ext3, which do not guarantee data integrity. Moreover, a 45.6% reduction in recovery time is achieved when compared with the full data journaling approach of ext3, and the memory overhead of VMA journaling is not significant.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to virtual machines and journaling file systems. Sections 3 and 4 describe the design and implementation of VMA journaling, respectively. Section 5 shows the performance results, which are followed by the discussion in Section 6. Finally, the related work and conclusions are given in Sections 7 and 8, respectively.

2. BACKGROUND

2.1. Virtual machines

Virtualization technology allows multiple isolated virtual machines to run concurrently in one physical machine, sharing the resources of that physical machine. Each virtual machine is supported by the VMM, a thin software layer running underneath the virtual machine. A virtual machine is also called a *guest domain*, or simply a *domain*, and the operating system running in a virtual machine is called a *guest operating system*. Generally, VMM can run on an operating system or an bare hardware. In this paper, we focus on the VMM running directly on bare hardware, which provides better performance and is commonly used in server environments.

VMM emulates the hardware interface and provides virtual computing resources, such as virtual CPU, memory and disk, to the guest operating systems running on the virtual machines. Virtual

DOI: 10.1002/spe

disks are usually implemented in the form of partitions or image files on the physical storage. The guest operating systems can utilize their preferred file systems to manage the virtual disks. Note that, strong isolation is supported among the virtual machines, so that a virtual machine crash cannot corrupt the other virtual machines or the VMM.

2.2. *Journaling file systems*

Journaling file systems adopt the concept of WAL to maintain file system consistency. In-memory file system updates are grouped into a transaction and then committed into the journal area, which is a reserved space on the storage, before they are flushed to the data area of the storage. When a system crashes, the file system can be efficiently brought back to a consistent state by replaying the *journal data* (i.e. data in the journal area).

Generally, journal commit occurs when the size of the current transaction exceeds the Maximum Transaction Size (MTS), a threshold defined by the file system. A larger value for the MTS threshold results in less frequent journal writes but may cause more data updates to be lost. By contrast, a smaller value for the threshold results in losing fewer data updates but incurs more frequent journal writes. Moreover, journal commit is also done periodically to ensure that each update will be committed to the journal area in a limited time. There are two common methods for journal commit. One is to write only the modified bytes into the journal area, called *logical journaling*, while the other is to write the complete blocks that contain the modified bytes into the journal area, called *physical journaling*. In this paper, we assume that physical journaling is used since the proposed technique is implemented in ext3, a file system that uses physical journaling.

To reclaim the space of the journal area, a checkpoint procedure has to be invoked to ensure that all the data updates corresponding to the to-be-reclaimed journal data have been flushed to the data area. The procedure checks the transactions in the journal area, flushes all the corresponding data that has not yet been written to the data area (i.e. the dirty buffers belonging to the transactions) and then reclaims the space used by those transactions until enough space has been reclaimed.

Many journaling file systems support more than one journaling modes with different performance and consistency strengths. For example, both ext3 and ReiserFS support three journaling modes: writeback, ordered and journal modes. The differences among these modes are the content in the journal area and the flush order of the data.

Both writeback and ordered modes log only metadata and therefore do not ensure data integrity. In the writeback mode, data are exposed to the buffer flushing threads of the operating system immediately after each file operation, and metadata are exposed after each journal commit. Since no flushing order is enforced in the writeback mode, metadata may be flushed back to the data storage before the associated data, causing the dangling pointer problem. Therefore, this mode is generally regarded as having the weakest consistency semantic among the three modes. In addition to ext3 and ReiserFS, the writeback mode is also supported by JFS and XFS. The ordered mode solves the dangling pointer problem by ensuring that the updated data are flushed back to the storage before the commit of the associated metadata. During a commit, dirty data buffers corresponding to the to-be-committed metadata are flushed to the data storage before the latter have been written to the journal area. Although the strict flushing order provides strong consistency semantic, the ordered mode does not guarantee data integrity since the data updates are not journaled, as in the writeback mode. For example, a system crash during the update of an in-file record could lead to an unrecoverable corruption of that record.

The journal mode supports full data journaling, that is, it logs both data and metadata and thus guarantees both file system consistency and data integrity. All the data and metadata updates are committed to the journal area before they are exposed to the buffer flushing threads of the operating system. However, this mode has inferior performance under most workloads for the following reasons. First, more writes are required in this mode since all the data and metadata updates have to be written to the storage twice. Second, journaling the data updates causes the commit and the checkpoint procedures to be triggered more frequently.

DOI: 10.1002/spe

3. DESIGN

In this section, we describe the design of VMA journaling. Section 3.1 presents the architecture overview. Section 3.2 describes the approach to ensure file system consistency and data integrity under VMA journaling. Journal data reclamation and file system recovery are described in Sections 3.3 and 3.4, respectively.

3.1. Architecture of VMA journaling

The major difference between the VMA journaling and traditional journaling approaches is the method of journal data handling. In order to eliminate journaling writes to the storage, VMA journaling commits journal information to the VMM instead of the storage. Figure 1 illustrates the difference between the handling of journal data in the traditional and VMA journaling approaches. In step 1, both traditional and VMA journaling groups dirty buffers, which reflect metadata and data updates, into a transaction. Then, the traditional journaling approach commits these dirty buffers to the on-storage journal area, as shown in step 2 of Figure 1(a). However, VMA journaling commits the dirty buffers to the journal area residing in the VMM memory instead of the storage, as shown in step 2 of Figure 1(b). Therefore, no journaling writes to the storage are needed. After the commit, the guest domain can flush the committed dirty buffers in an asynchronous manner, as shown in step 3 of Figures 1(a) and (b). When a domain crashes, the information in the journal area can be used for file system recovery.

One straightforward implementation of the proposed architecture is to move the journal area from the storage to the VMM memory. However, such an approach has the following two problems. First, it consumes a significant amount of VMM memory. A journaling file system usually reserves a fixed size journal area, typically between 32 and 128 MB, to store its journal data. If a VMM supports 10 virtual machines, each of which has a single journaling file system mounted as VMA journaling, and each journaling file system requires a 64 MB journal area, then the total memory requirement of the in-VMM journal areas would be 640 MB. Reducing the size of each in-VMM journal area reduces the memory requirement. However, a smaller journal area will be exhausted more quickly, causing the checkpoint procedure to be triggered more frequently, and thus degrading the performance [6]. Second, journaling file systems that use full data journaling, such as the journal mode of ext3, could commit a significant amount of data (including metadata) to the journal area. In this case, in-VMM journal area would lead to a huge amount of data being transferred from guest domains to the VMM, degrading the system performance. Moving the journal area from the storage to the guest domain memory seems to be an alternative. However, this faces a similar problem since each domain has to reserve a large memory space for the journal area(s),

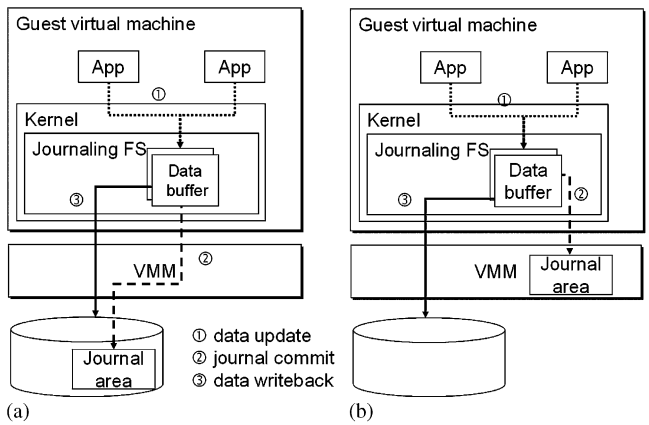

Figure 1. (a) Traditional journaling vs and (b) VMA journaling.

DOI: 10.1002/spe

causing more frequent memory pressure of the domain, and a large volume of data could be copied to the journal area when full data journaling is used.

VMA journaling avoids these problems by using a single copy of buffer to represent both the updated data and the journal data. In contrast to traditional journaling file systems that duplicate to-be-committed dirty buffers to the journal area, VMA journaling transfers only the information for locating the buffers to the VMM. Specifically, committing a buffer under VMA journaling involves transferring only the *journal information*, i.e. the tuple (*memory_address*, *block_number*) denoting the machine memory address and the corresponding disk block number of the buffer, to the VMM. As a result, a single copy of the buffer is used to represent both the updated data and the journal data. Once a domain crashes, VMA journaling allows the VMM to retain the memory of the committed dirty buffers, and it flushes the retained buffers back to the storage for maintaining file system consistency and data integrity.

It should be noted that the memory requirement of the in-VMM journal area is not significant since only the metadata for locating the journal data is stored in that area. Moreover, instead of statically reserving a fixed-size memory for each in-VMM journal area, all the memory used by the journal area is dynamically allocated. When memory pressure occurs in the VMM, the guest journaling file system that occupies the largest journal area size will be notified to reduce its journal area.

3.2. Ensuring file system consistency

As mentioned above, VMA journaling records both metadata and data updates. Therefore, just as in full data journaling, both file system consistency and data integrity are ensured. However, since the journal data are placed in the guest domain memory, unauthorized modifications to these data could disrupt the file system consistency and data integrity. For example, unintentional wild writes in the guest domain kernel might corrupt the journal data, leading the file system to an unrecoverable state.

To prevent wild writes from modifying the journal data, VMA journaling write-protects the memory pages of the to-be-committed dirty buffers when committing a transaction. Therefore, the guest operating system can detect wild writes to the journal data via page faults, and deny those writes. Note that this approach differs from the synchronous page protection approach (i.e. unprotecting/protecting the buffer right before/after each buffer update) used in RIO [33], in two aspects. First, VMA journaling protects the to-be-committed dirty buffers in batch. Synchronous page protection could easily lead to a large number of switches between the VMM and guest domains, and thus degrade the system performance, since page protection/unprotection involves modifying page table entries, which are managed by the VMM. By contrast, batch protection prevents frequently updated buffers from frequent protection/unprotection, thus resulting in a much lower overhead. Second, VMA journaling ensures file operation atomicity and data integrity. Atomicity is achieved by protecting the dirty buffers belonging to a transaction and recording the corresponding journal information in a single call to the VMM (i.e. a hypercall). VMM ensures that all or none of the buffers are protected and have their information recorded in the journal area. Moreover, since both data and metadata are committed atomically in units of a transaction, they can be used for ensuring data integrity after a domain crash. More detailed discussions between RIO and VMA journaling are provided in the Related Work.

Unlike a wild write, a write issued from the file system to a protected buffer should be allowed since the buffer still represents the updated data. Copy-on-write (COW) is used for such a write. That is, the content of the protected buffer is copied to a free and unprotected buffer, which is used for satisfying the write. On the next commit, these two buffer copies are merged by write-protecting the buffer containing the most up-to-date data and freeing the original protected buffer that contains the stale data. After the merge, a single copy of buffer again represents both the updated data and the journal data.

Figure 2 shows an example of this process. Before the commit of buffer page *A*, updates of that buffer are performed on the buffer page directly, as shown in step 1. At this time, *A* represents the updated data. Once *A* has been committed, as shown in step 2, it represents both the updated

DOI: 10.1002/spe

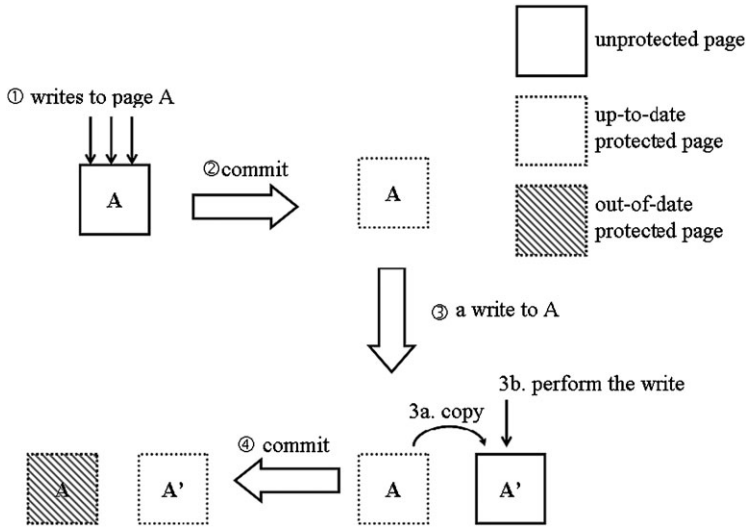

Figure 2. Example of journal commit and copy-on-write (COW).

data and the journal data. Therefore, it is write-protected to prevent data corruption from wild writes. In step 3, a further file system update to A triggers the COW operation, which copies the data of A to a free page A' and then performs the update on A'. After the page copying, A' represents the updated data while A still represents the up-to-date journal data. Thus, further file system updates can be performed on A' directly. In step 4, A' is committed and represents both the up-to-date journal data and the updated data, while A becomes the out-of-date journal data and can be released. Releasing out-of-date journal data does not compromise file system consistency since only up-to-date journal data are used for file system recovery.

Note that out-of-date journal data may have been written to the storage before being released since dirty buffers are exposed to the buffer flushing threads of the operating system when they are committed (i.e. become the journal data), as in full data journaling of ext3. For example, in Figure 2, A may have been written to the storage before the commit of A'. However, this does not compromise file system consistency since the file system can be recovered by writing the up-to-date journal data to the storage when the domain crashes. For example, in Figure 2, if the domain crashes right after the commit of A' (i.e. step 4), the data can be recovered by writing the content of A' to the corresponding block of the storage.

3.3. Reclamation of journal data and information

As mentioned before, in VMA journaling, a protected buffer represents both the updated data and the journal data. Reclamation of the journal data is triggered when the buffer is going to be reclaimed by the guest operating system, for instance, under memory pressure. When the buffer is going to be reclaimed, VMA journaling unprotects the buffer and removes the journal information corresponding to the buffer via a hypercall. After that, the buffer no longer represents the journal data and can be reclaimed by the guest operating system. Note that file system consistency and data integrity still remain since a buffer is reclaimed only after its content has been flushed to the data area.

However, unprotecting each buffer during its reclamation could cause a significant overhead due to frequent switches between the VMM and guest domains. To reduce the overhead, batch unprotection is adopted. Specifically, in addition to unprotecting the to-be-reclaimed buffer, VMA journaling also unprotects all the clean (i.e. non-dirty) and protected buffers in the committed transactions. Since those buffers have had their data flushed to the data area, unprotecting them raises no file system consistency problems. Figure 3 shows the pseudocode of journal data reclamation.

DOI: 10.1002/spe

```
reclaim_journal_data (victim page v)
/* v is a victim buffer page that is going to be reclaimed by the guest OS */
1   if v is not protected
2       return SUCCESS   /* v has been unprotected by prior batch unprotection */
3   end if
4   S := {v}          /* S is the set of buffers that need to be unprotected in batch */
5   L := list of buffers in the committed transactions
6   for each buffer B in the list L
7       if B is clean and protected
8           S := S U {B}
9       end if
10  end for
11  remove all buffers in S from list L
12  ret : = unprotect all buffers in S and remove the corresponding journal information
13  return ret
```

Figure 3. Algorithm of journal data reclamation.

Reclamation of the journal data and information also takes place when the VMM cannot afford the journal area of the guest domain. When this situation occurs, the checkpoint procedure is triggered, during which the journal data are written to the data area until the size of the journal area has been reduced to below a specific threshold. Nevertheless, this situation rarely occurs because the size of a journal area is quite small in VMA journaling. Specifically, VMA journaling only stores a journal information tuple for each page protected by it in the journal area and thus the size of a journal area S_{journal} can be expressed by the following equation:

$$S_{\text{journal}} = N_{\text{protected}} \cdot S_{\text{tuple}} + C \tag{1}$$

where $N_{\text{protected}}$ denotes the number of pages protected by VMA journaling, S_{tuple} denotes the size of a journal information tuple and C denotes the size of the data structures used to manage a journal area. Since $N_{\text{protected}}$ can never be larger than N_{pages}, the number of pages in the domain, the following inequality holds:

$$N_{\text{protected}} \leqq N_{\text{pages}} = M_{\text{domain}} / M_{\text{page}} \tag{2}$$

In (2), M_{domain} and M_{page} denote the memory sizes of a domain and a page, respectively. Therefore, from (1) and (2), the following holds:

$$S_{\text{journal}} \leqq (M_{\text{domain}} / M_{\text{page}}) \cdot S_{\text{tuple}} + C \tag{3}$$

In the current implementation, S_{tuple} and C are 12 and 512 bytes, respectively. Thus, for a guest domain with 1 GB of main memory and 4 kB pages, the maximum size of the journal area is only about 3 MB. Such a small size as well as the eager journal area reclamation resulting from batch unprotection cause the checkpoint procedure to be rarely triggered in VMA journaling. Furthermore, we can completely prevent checkpointing by simply reserving 3 MB of journal area for the domain mentioned in the example. The reserved space can be even less than 3 MB since not all of the main memory can be used by the file systems in a domain.

3.4. File system recovery

Similar to traditional journaling approaches, VMA journaling performs recovery by simply replaying the journal data. In VMA journaling, this is achieved through the cooperation of the VMM and the system management domain (e.g. domain 0 in the Xen virtualization environment). Note that, as mentioned in Section 3.2, some journal data may have been flushed to the data area. File system recovery involves writing the not-yet flushed journal data to the data area to ensure file system consistency and data integrity.

DOI: 10.1002/spe

When a guest domain crashes, the VMM reclaims all memory pages of the domain except those containing the journal data (i.e. the protected buffers). Then, the VMM notifies the system management domain, which wakes up a recovery thread to start the following recovery procedure. First, for each file system mounted as VMA journaling in the crashed domain, the recovery procedure issues a query for the total size of the journal data. If the size is zero, the corresponding file system is consistent and the recovery thread goes on to check the next file system. Otherwise, the recovery thread prepares a free memory pool of that size for exchanging with pages containing these journal data. It then issues a hypercall to perform the memory exchange to obtain the journal data and to retrieve the corresponding journal information from the VMM. After the exchange has been completed, the recovery thread writes the journal data back to the data area according to the journal information. The memory exchange is implemented by page remapping. The journal data are remapped into the system management domain, and the pages in the free memory pool are remapped into the VMM. After the journal data have been written, the memory containing the journal data becomes free memory of the system management domain, and the recovery thread informs the VMM to reclaim the journal information about the file system. Note that a threshold is set on the maximum size of the free memory pool. If the total size of the journal data is larger than the threshold, the steps are repeated until all the journal data have been written back. As in traditional journaling file systems, whole-storage scanning is not required.

Note that writing journal data to the data area can also be done without memory exchanges. For this, the memory pages containing the journal data can be remapped into the system management domain, written to the data area and then returned to the VMM. However, memory exchange is used in the current implementation since it allows the VMM to have the free memory without waiting for the journal data to be written back to the data storage. The free memory can be used to serve incoming VMM requests (e.g. creation of a new domain) immediately after the memory exchange.

4. IMPLEMENTATION

We implemented VMA journaling by modifying the Journaling Block Device (JBD) layer and the journaling-related functions of the ext3 file system (specifically, the journal mode of ext3) in Linux, resulting in a new journaling mode of ext3, called VMA mode. File system functions that are not related to journal area management, such as grouping of dirty buffers, block allocation and directory entry seeking, are intact. This assists in porting VMA journaling to other journaling file systems. In addition to the modifications in ext3 and JBD, we also augmented the Xen VMM to support VMA journaling and implemented the recovery procedure in the system management domain of Xen (i.e. domain 0). Note that the implementation of VMA journaling does not cause a noticeable increase in the complexity of Xen since only less than 600 lines of C code are added to Xen, allowing Xen to remain simple and stable. In the following, we describe the implementation details.

4.1. VMA journaling interface

A hypercall interface was implemented to support VMA journaling. Specifically, we added a new hypercall called *VM_journal()*, which can be used by VMA journaling aware file systems as well as by the recovery thread to handle the journal data. Table I lists the operations supported by the hypercall. The *LOG_OP_INIT* operation instructs the VMM to initialize a journal area. This operation is usually invoked during the mount of the file system, and parameters, such as the file system identifier, block size and device name, are specified upon the invocation of this operation. Note that the block size parameter is used to ensure that the size of a file system block is equal to the page size. In the current implementation, we support only file systems with block sizes equal to the page size. This makes the implementation easier. For example, partial-page writes may need to be issued by the recovery thread if the block size is smaller than a page.

The *LOG_OP_MPROTECT* operation is used to protect and unprotect journal data. Two sets of buffers can be specified when invoking the operation, the set of buffers that need to be protected

DOI: 10.1002/spe

Table I. Operations supported by the VM_journal() hypercall.

Journal operations	Description
LOG_OP_INIT	Register guest domain information and initialize a journal area
LOG_OP_CLEANUP	Clean the journal area stored in the VMM
LOG_OP_MPROTECT	Modify protection status of the buffers: for buffers that need to be protected, record the journal information and protect the buffers; for buffers that need to be unprotected, un-protect the buffers and remove the corresponding journal information
LOG_OP_QUERY	Retrieve the total size of the journal data of the given file system
LOG_OP_RECOVER	Retrieve journal information from the VMM and exchange the given memory pool with the journal data

and the set of buffers that need to be unprotected. The *LOG_OP_QUERY* operation is invoked by the recovery thread to retrieve the total size of the journal data corresponding to the given file system. The *LOG_OP_RECOVER* operation is used to perform the memory exchange mentioned in Section 3.4 and to retrieve the journal information from the VMM. Finally, the *LOG_OP_CLEANUP* operation is used to remove the journal area in the VMM when unmounting the file system or during the execution of the recovery procedure.

4.2. Write access checking

Before updating a buffer, a journaling file system must invoke its write access checking code to verify that the buffer is updatable. For example, in ext3, a buffer cannot be updated directly if it belongs to an older transaction or it is under committing. We extend the write access checking procedure of ext3 (i.e. the *get_write_access()* function) to handle writes to protected buffers.

As mentioned in Section 3.2, COW is used for handling writes to protected buffers. We do not reuse the COW code of ext3, however, since it requires a pair of extra protection and unprotection operations for each protected buffer, leading to a larger overhead. Specifically, the COW scheme of ext3 makes a copy of the buffer and then updates the original copy. Therefore, it has to unprotect the original copy and protect the new one. To eliminate such overhead, we apply the update on the new copy instead of the original one, which is achieved by external reference redirection after buffer copying. That is, the external references to the original buffer are redirected to the new one so that further updates are performed on the latter. In our work, 13 functions in ext3 were patched to implement the external reference redirection. Figure 4 illustrates how the code patch works when the file system modifies a directory entry residing in a protected page, which represents a buffer and hence is referred by a buffer data pointer (specifically, the *b_data* field of structure *buffer_head*). In step 1, a pointer *dir_entry* in the file system code refers to the target entry that needs to be modified. To redirect the *dir_entry* pointer, the patched code saves the page offset k of the *dir_entry*. In step 2, the content of the protected page is copied to a newly allocated page (i.e. the *cow_page*) and then the buffer data pointer and the related kernel references such as the radix tree and the LRU list of the page cache are redirected to the new page. Finally, in step 3, the new value of the *dir_entry* is obtained by adding the new value of the buffer data pointer and the offset k. According to Figure 4, steps 1 and 3 require the file system code to be patched. On the next commit, the original buffer page contains out-of-date data and thus is called the Out-of-Date Protected (ODP) buffer. All the ODP buffers are useless and they are unprotected and released in batch.

4.3. Transaction commit and journal data reclamation

The commit procedure of VMA journaling is triggered periodically, upon file system synchronization or when the total size of buffers belonging to the current transaction exceeds the MTS. The MTS and the interval between successive periodic commits are set to 32 MB and 5 s, respectively, in the current implementation, the same as the default setting of ext3 on a disk partition larger than 4 GB.

DOI: 10.1002/spe

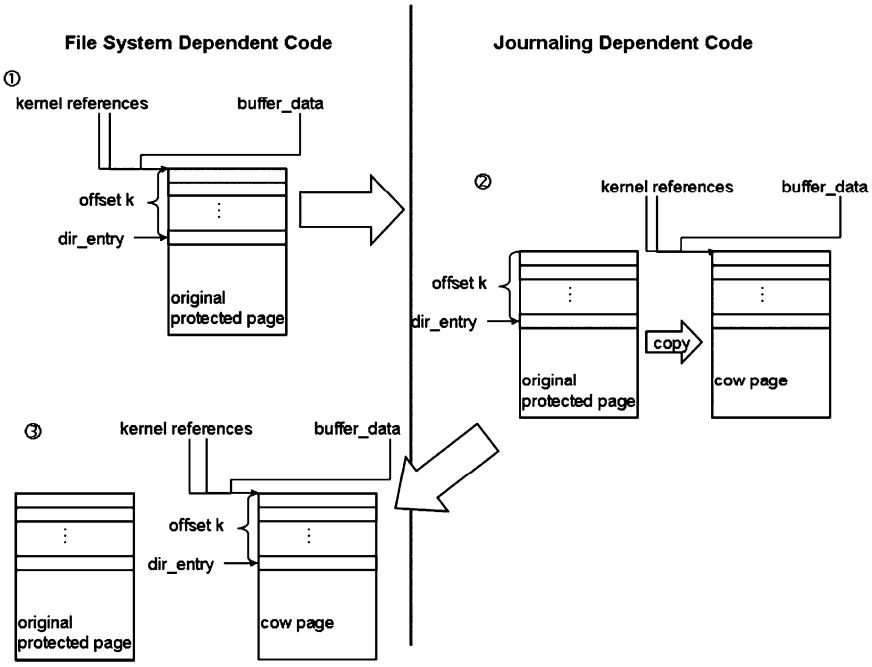

Figure 4. Example of external reference redirection.

The commit procedure is as follows. First, the information corresponding to the dirty buffers in the current transaction is gathered. Second, the *LOG_OP_MPROTECT* operation is invoked to protect these buffers and to unprotect the ODP buffers in batch. Third, all the buffers protected in the second step are inserted into the tail of the committed buffer list, which will be described later. Finally, the buffers are exposed to the dirty buffer flushing threads of the guest operating system so that they can be flushed back to the storage later.

Note that, although the procedure of committing dirty buffers in VMA journaling differs from that in traditional journaling approaches, the procedures of *grouping* dirty buffers are the same in both VMA journaling and traditional journaling approaches. In the current implementation, VMA journaling reuses the code of the journal mode of ext3 to group dirty buffers corresponding to metadata and data updates. As a result, porting VMA journaling to other journaling file systems does not require re-implementing the code of grouping dirty buffers, reducing the porting effort.

As mentioned above, batch unprotection is used for journal data reclamation. Specifically, when a protected buffer is going to be reclaimed, VMA journaling unprotects the buffer as well as all the clean buffers in the committed transactions in batch via a single hypercall. Since all the buffers belonging to the committed transactions are inserted into the committed buffer list, the clean buffers can be found by simply scanning this list. In order to reduce the scanning time, we move a buffer to the head of the list when it becomes clean. As a consequence, scanning can be terminated when a dirty buffer is encountered.

4.4. Journal information management

As mentioned in Section 3.1, all the space required by the in-VMM journal areas is allocated on demand. Each journaling file system instance has an associated data structure called VMA journaling control block (*VM_JCB*), which contains the file system and domain-specific data (e.g. block size, domain identifier, file system identifier) and a hash table of the journal information. A *VM_JCB* structure is created when the *LOG_OP_INIT* operation is invoked (during the mount of a guest file system with VMA mode) and is removed when the *LOG_OP_CLEANUP* operation is invoked. All the *VM_JCB* structures are chained in a list called *JCB_list*.

DOI: 10.1002/spe

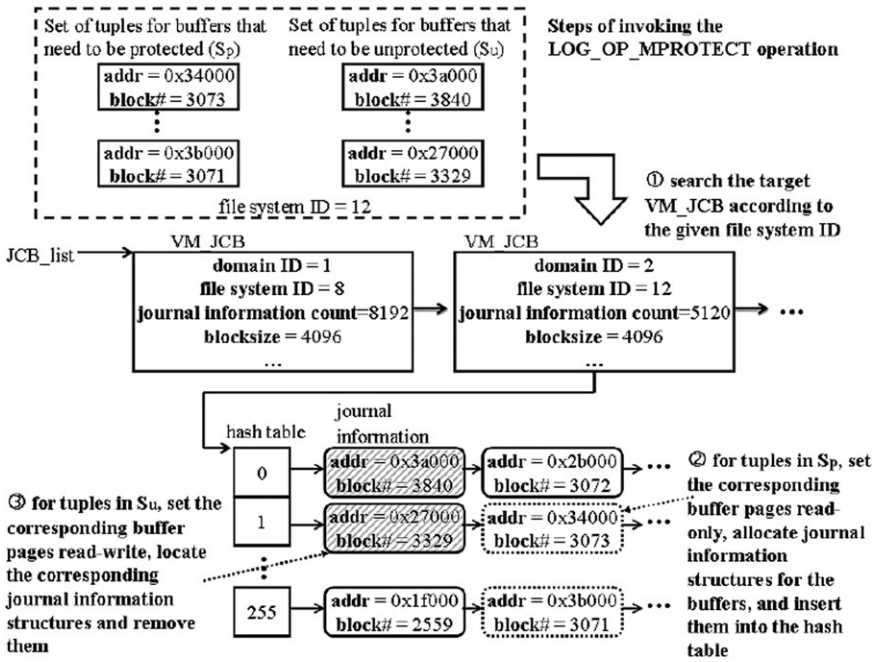

Figure 5. Journal information management.

When the *LOG_OP_MPROTECT* operation is invoked, as shown in Figure 5, the *VM_JCB* structure corresponding to the guest file system is first located according to the given file system identifier (step 1). Then, the journal information tuples corresponding to the buffers that need to be protected are inserted into the hash table of the *VM_JCB* structure after the buffers are protected (step 2). Finally, all the tuples corresponding to the buffers that need to be unprotected are removed from the hash table (step 3).

5. PERFORMANCE EVALUATION

In this section, the performance of VMA journaling is evaluated. Section 5.1 describes the experimental environment and the benchmarks used for performance evaluation. Section 5.2 presents the performance results under these benchmarks. The comparison of the time of file system recovery is given in Section 5.3. The memory overhead of VMA journaling is presented in Section 5.4. Section 5.5 demonstrates that the VMA mode ensures data integrity, whereas the metadata journaling modes of ext3 do not. Finally, Section 5.6 presents the code and data sizes of the current implementation of VMA journaling.

5.1. Experimental environment

Table II shows the machine configuration and the benchmarks used for performance evaluation. Up to eight guest domains were run on a physical machine, with 128 MB of memory allocated in each domain. In order to prevent system I/O activities from affecting the performance results, two virtual disks belonging to two separated physical disks were allocated for each domain. The virtual system disks stored the program images including the operating systems, libraries and benchmarks, while the virtual data disks stored the data accessed by the benchmarks. Each virtual disk was made up of a single partition on the corresponding physical disk, and the data access performance of the virtual data disks was reported.

Three micro-benchmarks, *seq_write*, *r nd_write* and *file_delete*, adopted from *filebench* [34] and three macro-benchmarks, *postmark* [35], *untar* and *kernel_compile*, were used for performance

DOI: 10.1002/spe

Table II. Experimental environment.

Hardware	CPU	Pentium 4—3.2 GHz
	Memory	DDRII 2 GB
	Disks	*System disk*: Western Digital WD800JD-22LS, 80 GB *Data disk*: Seagate ST336753LW, 1.5K RPM, 36.7 GB, 4-ms seek time, 2-ms rotational delay, transfer speed: 63 MB/s (avg.)
VMM	Xen 2.0.7	
Domains	Kernel	XenoLinux 2.6.11
	Memory	128 MB for each guest domain 256 MB for domain 0
	Virtual disks	*Virtual system disk*: 8 GB *Virtual data disk*: 4 GB
Micro benchmarks	Filebench: *seq_write*, *rnd_write* and *file_delete*	
Macro benchmarks	*Postmark*, *untar* and *kernel_compile*	

evaluation. In the *seq_write* benchmark, a single empty file is first created and then has 8 kB of data appended each time until the file size reaches 600 MB. This benchmark is data write intensive and is used to evaluate the benefit of journal write elimination in VMA journaling. The *rnd_write* benchmark issues a sequence of 4 kB random writes to a 512 MB file until 128 MB of data have been written. This benchmark is used to evaluate the performance of random writes. The *file_delete* benchmark first allocates 40 K files in 100 directories (i.e. the file creation stage) and then uses 16 threads to delete these files (i.e. the file deletion stage). The file size is set as a gamma distribution with the median size of 16 kB. Note that since the goal of this benchmark is to evaluate the performance of file deletion, only the time spent in the file deletion stage is measured. File deletion causes the operating system to immediately reclaim the buffers containing the file data, which in turn causes buffer unprotection in VMA journaling. Therefore, this benchmark is used to measure the performance impact of frequent buffer unprotection. *Postmark* simulates the workload of a news or email server. During execution, it creates an initial set of small files and then applies a number of transactions on those files. Each transaction consists of a create/delete operation together with a read/append operation. Thus, it is a metadata access dominated and I/O-intensive workload. In the experiment, the initial file set contains 10 K files residing in 200 directories, the file size ranges from 0.5 to 9.8 kB (i.e. the default setting of *postmark*), and 25 000 transactions are performed. The *untar* benchmark extracts a Linux source tree (version 2.6.11) from a bzip2-compressed image. The size of the compressed image is 36 MB and the decompressed source tree includes 17 090 files, ranging from 6 bytes to 853 kbytes, in 1083 directories and the total size is 231 MB. The *kernel_compile* benchmark is a CPU-intensive workload, which builds a compressed kernel image from the source tree of the Linux kernel (version 2.6.15). The total size of the object files and the kernel image is about 25 MB.

5.2. Performance results

In this section, we present the performance results of the VMA mode and the other journaling modes of ext3 under the benchmarks. In each experiment, the average execution time (with standard deviation) of 10 iterations of benchmark execution is reported. In each iteration, various number of domains were run concurrently, on each of which an instance of the given benchmark was executed (with cold cache) and the average execution time was recorded. The execution time was measured by the benchmarks (i.e. *filebench* or *postmark*) or the GNU *time* command. Note that all the performance evaluations, except for that in Section 5.2.6, were performed on initially empty virtual disks. Section 5.2.6 shows the results on non-empty virtual disks.

In addition to measuring the performance of all the journaling modes, we also implemented a modified version of ext3 by directly removing its journal writes and measured the performance results of this modified version. Although this modified version cannot ensure file system

DOI: 10.1002/spe

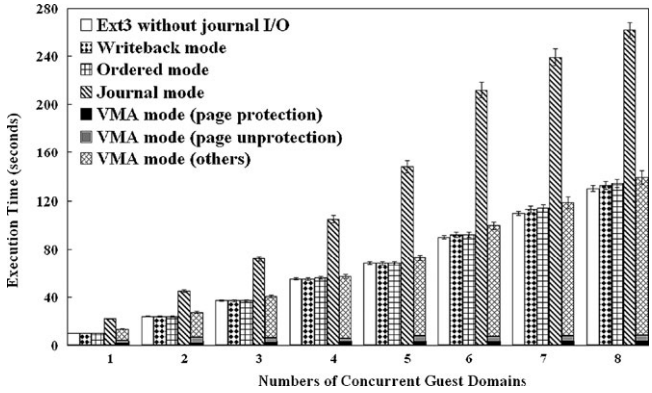

Figure 6. Performance of *seq_write* under concurrent domains.

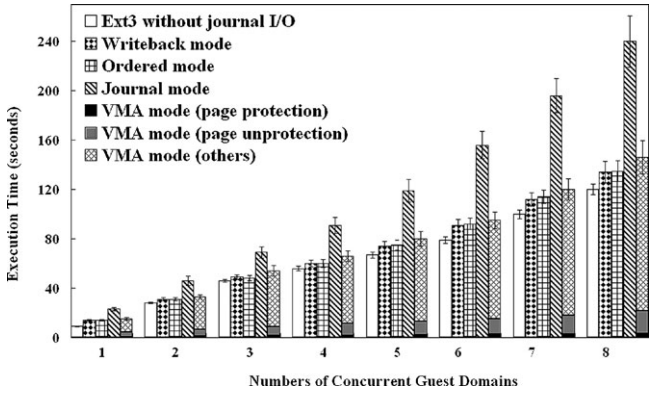

Figure 7. Performance of *rnd_write* under concurrent domains.

consistency, its performance results can serve as the performance upper bound for all the journaling modes.

5.2.1. Results under micro benchmarks. Figure 6 shows the results of the *seq_write* benchmark under one to eight concurrent domains. Standard deviations are shown in error bars. The time spent in buffer protection/unprotection is also shown in the results of the VMA mode. From the figure, it can be seen that the VMA mode outperforms the journal mode by up to 50.9%, which is due to the elimination of journal writes. Moreover, VMA mode achieves similar performance with the metadata journaling modes, which do not guarantee data integrity. For example, in the case of eight domains, VMA journaling increases the average execution time by only 5 and 3.4% compared to the writeback and ordered modes, respectively. The major overhead of VMA journaling comes from the page protection/unprotection. Figure 7 shows the results of the *rnd_write* benchmark. Similar to the results in Figure 6, the VMA mode outperforms the journal mode (by up to 38%) due to the elimination of journal writes, and it achieves similar performance with metadata journaling modes (with only 3.3–9.8% increase in the average execution time). However, the portion of time spent on page unprotection in the *rnd_write* benchmark is larger than that in the *seq_write* benchmark. This comes from the slower flushing speed of dirty buffers, which results from the lower performance of random disk writes when compared with sequential writes. As mentioned in Section 3.3, the batch unprotection technique of VMA journaling unprotects all the clean and protected buffers in a single hypercall to reduce the performance impact of switches between the domains and the VMM. With slower flushing of dirty buffers in the *rnd_write* benchmark, fewer buffer pages are unprotected in a page unprotection hypercall, increasing the invocation frequency of that hypercall.

DOI: 10.1002/spe

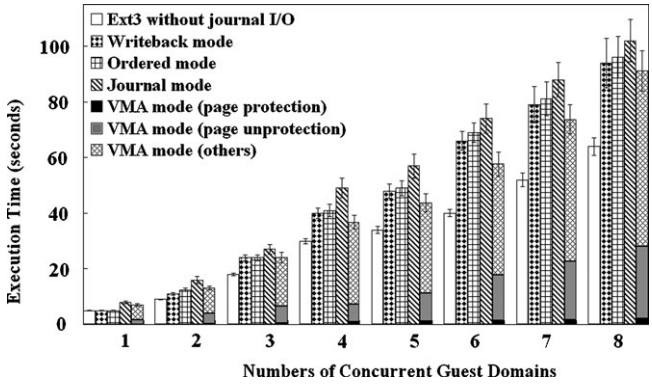

Figure 8. Performance of *file_delete* under concurrent domains.

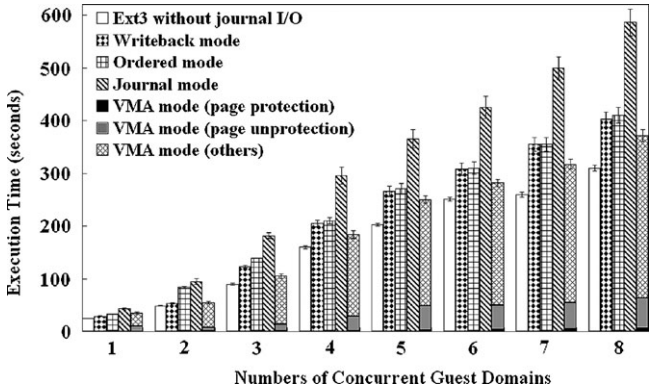

Figure 9. Performance of *postmark* under concurrent domains.

Figure 8 shows the results of the *file_delete* benchmark. As mentioned in Section 5.1, only the time spent in the file deletion stage is measured. The figure indicates that the VMA mode outperforms the journal mode by 10.8–24.6%. Moreover, since the benchmark is dominated by metadata writes, VMA mode even outperforms the writeback and ordered modes, by up to 16.4%, when the number of domains is larger than 2. Note that, page unprotection contributes to a relatively large portion of the execution time in this benchmark. This is because deleting a file triggers immediate unprotection of the pages containing the content of that file, increasing the invocation frequency of the hypercall for page unprotection. The cost in page protection/unprotection hypercalls is about 20–31% of the overall execution time, most of which comes from page unprotection resulting from file deletion. However, the cost is usually outweighed by journal write elimination and thus VMA mode still achieves the best performance among the four journaling modes in most of the cases under this benchmark.

5.2.2. Results under macro benchmarks. This section presents the performance results under the macro benchmarks: *postmark*, *untar* and *kernel_compile*. Figure 9 shows the results of *postmark* under one to eight concurrent domains. As shown in the figure, VMA mode achieves the best performance among the four journaling modes in almost all the cases under this benchmark. Specifically, it outperforms the journal mode and the metadata journaling modes by up to 42.3 and 24.5%, respectively, under multiple concurrent domains. Owing to the elimination of the journal writes, VMA mode reduces the write traffic to the storage by about 50% compared to the journal mode and by about 8% compared to the metadata journaling modes. The cost of page protection/unprotection contributes 13.3–30.4% of the overall execution time and most of the cost

DOI: 10.1002/spe

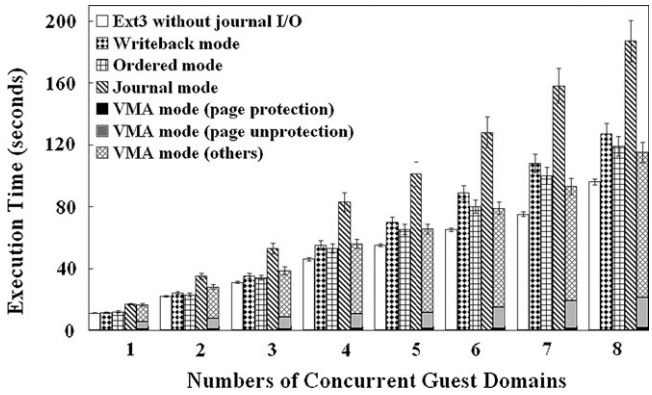

Figure 10. Performance of *untar* under concurrent domains.

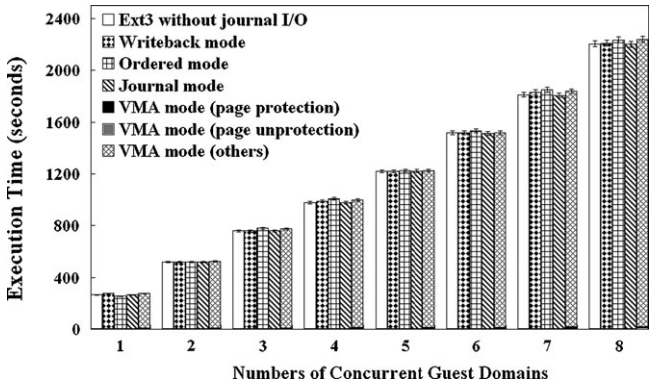

Figure 11. Performance of *kernel_compile* under concurrent domains.

comes from page unprotection. According to our measurement, 92% of the page unprotection overhead results from file deletion in *postmark*.

Figure 10 shows the results of *untar*. As shown in the figure, the VMA mode outperforms the journal mode by up to 41.1%, which is due to the reduction of write traffic to the storage by up to 49%. However, the VMA mode does not result in performance superior to the metadata journaling modes when there are four or fewer domains. This is because the *untar* benchmark generates fewer metadata writes when compared with *postmark*. In spite of this, VMA journaling ensures data integrity, and its performance is superior to that of metadata journaling modes when the number of concurrent domains is larger than 4. Figure 11 shows the results of the *kernel_compile* benchmark. As expected, all the modes show similar performance since the workload is CPU-bound. The cost of page protection/unprotection is not noticeable due to the limited number of invocations to the corresponding hypercall.

5.2.3. Performance vs data loss. As mentioned in Section 3.2, to avoid large performance degradation resulting from synchronous page protection, VMA journaling protects to-be-committed dirty buffers in batch when committing a transaction. However, this causes not-yet-committed data updates to be lost if there is a domain crash. As in traditional journaling file systems, the maximum size of lost data updates is bounded by the MTS, a tunable parameter in a journaling file system. In VMA journaling, a larger MTS value results in fewer invocations to the page protection hypercall but may cause more data updates to be lost. In contrast, a smaller MTS value results in losing fewer data updates but incurs more frequent page protection. Figure 12 shows the performance of VMA journaling with different values of MTS under *postmark* and *untar*. In the experiment, a single guest domain is used. The results are normalized to that with MTS as 32 MB, the default

DOI: 10.1002/spe

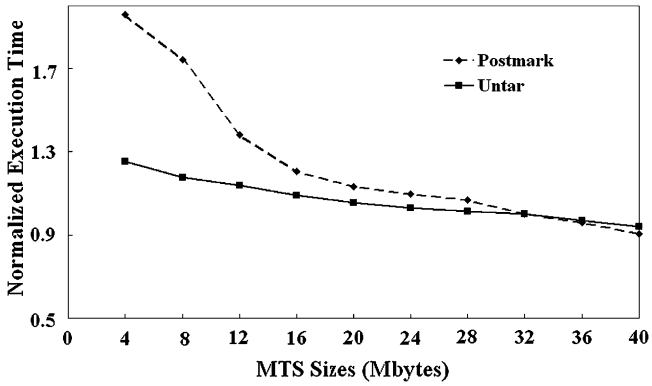

Figure 12. Performance of VMA journaling under different values of MTS.

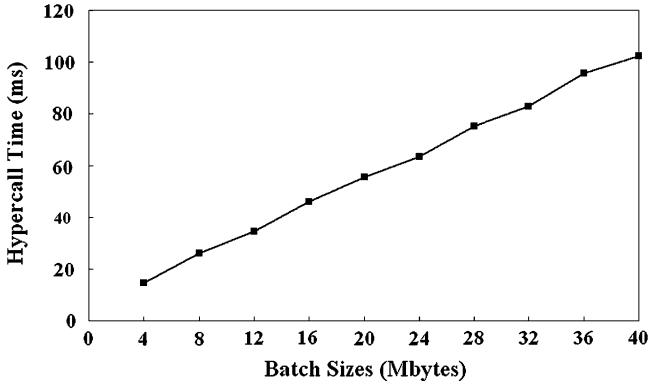

Figure 13. Execution time of the *LOG_OP_MPROTECT* operation with different batch sizes.

setting used in this paper. As expected, performance improves with the growth of MTS. As in traditional journaling file systems, in VMA journaling, the value of MTS can be chosen, when mounting the file system, by the system administrator according to the tradeoff between system performance and amount of data loss.

5.2.4. Execution time of VMA journaling hypercall. Since each invocation to the VMA journaling hypercall (i.e. *VM_journal()*) causes a synchronous trap from the guest domain to the VMM, understanding the latency of this hypercall helps us to clarify the overhead of VMA journaling. Among the operations of the *VM_journal()* hypercall, *LOG_OP_MPROTECT* has a larger impact on the system performance since it is invoked much more frequently than the other operations. Figure 13 shows the time required to protect different sizes of journal data in batch, through the *LOG_OP_MPROTECT* operation, under a single guest domain. Not surprisingly, the time increases with the growth of the data size. More importantly, the average time to protect a page decreases with the increase in the batch size, demonstrating the benefit of batch protection. For example, the time for protecting a single page decreases from 14.2 to 10 μs (i.e. a 29.6 % reduction) when the batch size grows from 4 to 40 MB. The time to unprotect journal data is similar to that for data protection and thus the results are not shown in this paper. The execution times of the other operations are all less than 20 μs.

5.2.5. Cost of checkpointing. Checkpointing is required to reclaim the space of on-storage or in-VMM journal area. As mentioned in Section 2.2, the checkpoint procedure checks the transactions in the journal area, flushes all the corresponding data that have not yet been written to the data

DOI: 10.1002/spe

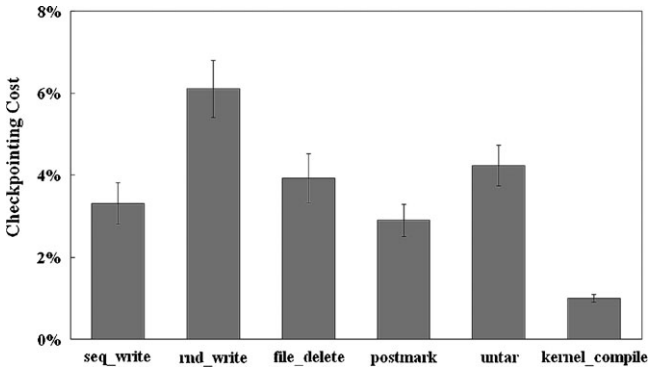

Figure 14. Cost of checkpointing in the journal mode of ext3.

area (i.e. the dirty buffers belonging to the transactions), and then reclaims the space used by those transactions until enough space has been reclaimed.

To measure the cost of checkpointing for a given journaling mode, the number of checkpointing events that occur during the execution of each benchmark under that mode is first recorded. If the number is not zero, the checkpointing cost is obtained by measuring the performance difference between that mode under a default-sized journal area and a large-sized journal area. The performance difference (i.e. the difference in the benchmark execution time) is then normalized to the benchmark execution time of that mode under the default-sized journal area. The size of the large-sized journal area is selected so that checkpointing never occurs under that size.

According to our results, checkpointing occurs only in the journal mode because all the metadata and data updates are stored in the journal area in that mode. Figure 14 shows the checkpointing cost of the journal mode under eight concurrent domains, with 128 MB default-sized journal area and 1 GB large-sized journal area. It can be seen that the average cost ranges from 1 to 6.2%. Although VMA journaling also logs all the metadata and data updates, only the journal information (i.e. the metadata for locating the journal data) is stored in the in-VMM journal area, consuming little memory space and thus never triggering checkpointing in the experiments.

5.2.6. Results on non-empty virtual disks. In this section, we show the performance of VMA journaling on aged and non-empty virtual disks. Before the execution of each benchmark, the layout of the virtual disk is initialized by *Impressions*, a framework for generating realistic file system images [36]. The disk space utilization is set as 77%, so that the utilization can be larger than 80% during the execution of each benchmark. Moreover, the parameter *layout score* is set as 0.1, so that on average only 10% of the blocks in a file are adjacent, to reflect a fragmented file system.

Figure 15 shows the performance of *postmark* and *untar* on empty and non-empty virtual disks, respectively. In the experiment, eight concurrent domains are run. As indicated by the figure, although performance on non-empty disks degrades for all the journaling modes due to file fragmentation, the performance differences among the modes are similar in empty and non-empty disks, showing that the effectiveness of VMA journaling remains on non-empty disks. The results of the other benchmarks are not shown since aging the disk has little performance impact under those benchmarks.

5.2.7. VMA journaling vs RAM disk-based journaling. As mentioned in Section 3.1, an alternative way to eliminate journal writes is to move the journal area from the storage to the guest domain memory. Figure 16 shows the performance comparison among traditional (i.e. magnetic disk-based) journaling, VMA journaling and RAM disk-based journaling. The latter is the same as traditional journaling of ext3 except that a 40 MB RAM disk is reserved from the guest domain memory as the journal area. Eight concurrent domains are used in this experiment. From the figure, writing journal data to a RAM disk results in worse performance than VMA journaling under all the benchmarks.

DOI: 10.1002/spe

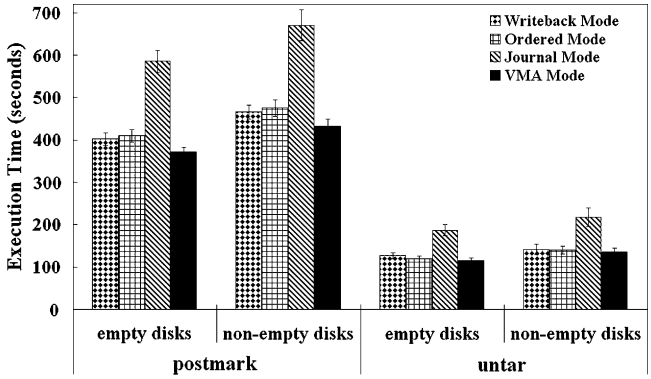

Figure 15. Performance of *postmark* and *untar* on empty and non-empty virtual disks.

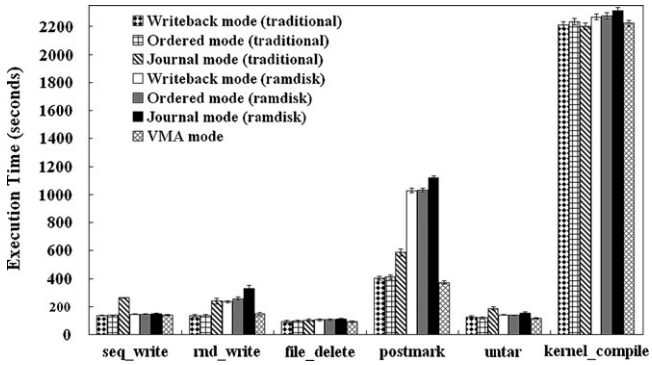

Figure 16. Performance comparison among VMA journaling, traditional (magnetic disk based) journaling and RAM disk-based journaling.

Table III. Recovery time and IO traffic during recovery.

		Writeback mode	Ordered mode	Journal mode	VMA mode
Postmark	Recovery time (s)	0.28±0.09	0.27±0.06	5.39±1.34	2.93±0.86
	IO traffic during recovery (MB)	9.3±2.2	9.1±1.6	132.2±30.8	69.6±18.4
Untar	Recovery time (s)	0.33±0.05	0.33±0.09	5.91±1.8	3.49±1.04
	IO traffic during recovery (MB)	12.9±1.8	12.9±2.2	130.8±31.2	68.8±18.5
Kernel_compile	Recovery time (s)	0.44±0.05	0.46±0.04	1.1±0.22	0.69±0.19
	IO traffic during recovery (MB)	16.5±1.5	16.5±1.8	33.6±7.6	24.3±3.7

Furthermore, it even has worse performance than traditional journaling under *rnd_write*, *postmark* and *kernel_compile*. This is mainly because a significant portion of the domain memory has to be reserved for the journal area, causing memory pressure of the domain more frequently.

5.3. Recovery time

This section evaluates the performance of file system recovery under different journaling modes. We reset a guest domain running the *postmark*, *untar* or *kernel_compile* benchmark when the benchmark has run for half of its execution time. Table III presents the recovery time as well as the amount of I/O traffic during the recovery procedure under each mode. Standard deviations are also shown in the table. The table shows that full data journaling modes have a longer recovery time than metadata journaling modes. This is because the former has to replay the committed data updates, resulting in more I/O traffic during the recovery. When compared with the journal

DOI: 10.1002/spe

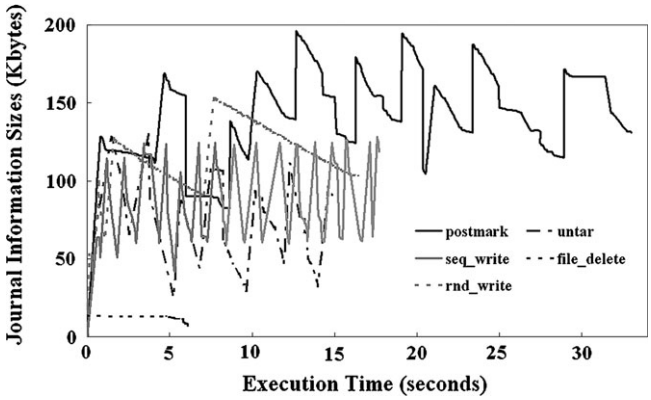

Figure 17. In-VMM memory overhead of VMA journaling.

mode, less I/O traffic is required in the VMA mode during the recovery. The is because the former scans the on-storage journal area for replaying the committed metadata and data updates, whereas the latter only flushes the dirty and protected buffer pages (i.e. the committed metadata and data updates that have not been flushed) to the storage. As shown in the table, this allows the VMA mode to reduce the recovery time by 45.6, 40.9 and 37.3% under *postmark*, *untar* and *kernel_compile*, respectively.

5.4. Memory overhead

Figure 17 shows the in-VMM memory overhead (i.e. the journal area size) of VMA journaling measured during the execution of a single domain. As shown in the figure, the memory overhead is less than 200 kB, demonstrating that the overhead is not significant. The rising edges in the figure correspond to batch protection while the falling edges correspond to batch unprotection. Many edges are steep, indicating that a large number of buffer pages can be protected/unprotected in batch. An exception occurs in the results of the *file_delete* benchmark, in which the size of the journal information drops slowly, meaning that only a small number of buffer pages can be unprotected in each page unprotection hypercall. As mentioned in Section 5.2.1, this is due to immediate buffer unprotection triggered by file deletion. Note that, under the *file_delete* benchmark, the size of journal information is not zero at time 0. This is because only the results of the file deletion stage, which follows the file creation stage, are shown. According to our measurement, up to 8192 and 8577 pages are protected and unprotected, respectively, in a single hypercall during the execution of the benchmarks. The frequencies for batch protection and unprotection are 0.2 to 1.2 and 3.8 to 891.9 times per second, respectively, under the benchmarks.

Next, the memory overhead in the guest domain, that is, the size of the extra memory resulting from COW, is measured. Note that COW increases the memory usage and could result in more frequent buffer replacement when the increased memory usage leads to memory pressure. Figure 18 shows the sizes of COW memory during the execution of a single domain running *postmark*. Since *postmark* is dominated by metadata writes, a large volume of protected metadata is duplicated to perform further file operations. As shown in the figure, the maximum overhead is about 10.3 MB. Note that, the COW overhead drops periodically since COW memory generated in a transaction will be reclaimed during the commit of that transaction. Moreover, the other benchmarks in this paper show negligible overhead (i.e. less than 100 kB in *untar* and less than 20 kB in the three micro benchmarks). The insignificant memory overhead would not lead to noticeable increase in the frequency of buffer replacement.

5.5. Data integrity

This section compares the journaling modes according to the level of data integrity. To achieve this, we execute a workload of updating records in a database file, crash the domain during the

DOI: 10.1002/spe

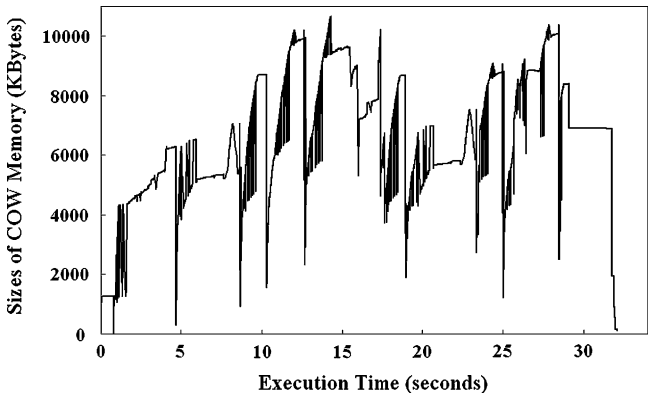

Figure 18. Memory overhead of copy-on-write in VMA journaling (*postmark*).

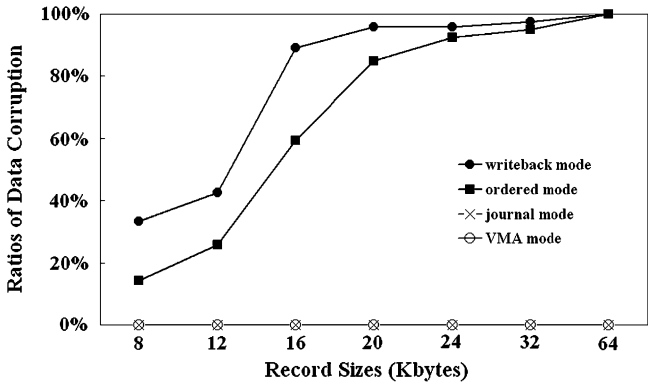

Figure 19. Ratios of data corruption.

execution of the workload and then check the integrity of the records after the file system recovery. Each record update is implemented by a *write()* system call, and an injected fault is triggered to crash the domain when 4 MB of data have been submitted to the I/O subsystems of that domain. Since a record may consist of multiple disk blocks, it could become corrupted if it is updated partially (i.e. some blocks contain new data while the other blocks contain old data). After the recovery, all the records are verified to check if there are corrupted records.

Figure 19 shows the results. The X-axis represents the record sizes. For each combination of record size and journaling mode, 120 iterations of the above experiment are performed and the ratios of the number of iterations with corrupted record(s) to the total number of iterations (i.e. 120) are shown (in the Y-axis). A single guest domain is used. From the figure, data integrity is ensured in journal and VMA modes. This is due to the logging of data updates under these two modes. For the writeback and ordered modes, which do not log data updates, the possibility of data corruption increases with the growth of the record size since partial updates of records occur more frequently under large records.

5.6. Code sizes

Table IV shows the code and data sizes of the Xen VMM and the ext3 file system, respectively, before and after the implementation of VMA journaling. The sizes of the recovery thread are also shown. From the table, the current implementation causes only small increases in the code and data sizes. Specifically, it increases the code and data sizes of the VMM by 7.2 and 0.1 kB, respectively, and it adds 4.2 and 0.7 kB to the code and data of ext3, respectively. The code of the recovery thread (in the system management domain) is 3.0 kB while the data size is only 0.7 kB.

DOI: 10.1002/spe

Table IV. Sizes of code and data before/after implementing VMA journaling.

	VMM		ext3		Recovery thread	
	Before code mod.	After code mod.	Before code mod.	After code mod.	Before code mod.	After code mod.
Code sizes (kB)	241.4	248.6	29.5	33.7	N/A	3.0
Data sizes (kB)	19.4	19.5	7.3	8.0	N/A	0.7

6. DISCUSSION

6.1. Reliability of hardware and virtual machine monitors

Although VMA journaling ensures data integrity and shows superior performance to full data journaling mode, it is not intended to replace the traditional journaling approaches in all kinds of virtualization environments. As mentioned before, VMA journaling is based on the assumptions that hardware errors do not lead to loss of critical data and sudden VMM crashes do not occur. The former assumption is similar to the one made in traditional journaling file systems that no faults in the storage subsystem lead to data loss [12]. Hardware subsystems are usually reliable in server platforms due to numerous techniques, such as redundant power, memory mirroring and RAID. For example, the annual failure rate of a disk drive is about 1.7–8.6% according to the previous study [37]. For drives with average annual failure rate of 4%, the probability of data loss is 0.16% for a 5-drive RAID 5 disk array. The probability can be even lower if RAID 10 is used. For another example, the average failure rate of a DIMM (Dual In-line Memory Module) is about 0.22% per year [17]. With memory mirroring, which is common in server platforms, the failure rate of a DIMM pair can be reduced to 0.0044% (i.e. $0.22\% \times 0.22\%$). For a server machine with four pairs of DIMMs, the probability of data loss is about 0.017%. This probability can be reduced further by using stronger ECC to recover more error bits or by utilizing proactive approaches to avoid data loss from DIMM failures (e.g. migrating critical data out of a DIMM when frequent correctable errors occur on that DIMM). The latter assumption is supported by the low complexity of VMMs. As mentioned in the Introduction, a micro-kernel seL4 [29], which can be used directly as a VMM, has been proved functionally correct, meaning that the kernel/VMM never crashes, falls into unknown states or performs unsafe operations. Therefore, bug-free VMMs are currently technically feasible [31]. Moreover, most VMM vendors are currently making efforts on improving the reliability of their VMM implementations.

Owing to these assumptions, VMA journaling targets at server platforms with reliable VMM and hardware subsystems and aims to ensure file system consistency and data integrity in the case of virtual machine crashes. Note that, VMA journaling is not the only work that assumes memory to be reliable. Previous studies have already demonstrated that memory can serve as reliable storage [33, 38–40], provided that fault tolerant techniques are used. These techniques are common in today's servers. Nevertheless, since the assumptions made by VMA journaling are stronger than that made by traditional journaling file systems, VMA journaling is not intended to replace the traditional journaling approaches in all kinds of virtualization environments. For example, traditional journaling modes may be preferable in a desktop platform without UPS since VMA journaling cannot ensure file system consistency and data integrity upon sudden power outage.

6.2. Applying VMA journaling to file systems with metadata journaling

VMA journaling ensures file system consistency and data integrity since it is implemented by applying the proposed approach mentioned in Sections 3 and 4 to the full data journaling mode (i.e. the journal mode) of ext3. With reduced journaling overhead in the proposed approach, full data journaling, which ensures data integrity, becomes less expensive and thus is used in VMA

DOI: 10.1002/spe

journaling. However, the proposed approach is not limited to be applied to file systems with full data journaling.

The major difference between full data journaling and metadata journaling is that the former groups updates of both data and metadata, whereas the latter groups metadata updates only. The proposed approach can be applied to a metadata journaling file system or a full data journaling one since it does not implement the code of grouping file system updates itself. Instead, as mentioned in Section 4.3, it reuses the code of the original journaling file system to group the updates. Therefore, the proposed approach can also be applied to a metadata journaling file system. In that case, only metadata updates are grouped and thus data integrity cannot be ensured.

From Sections 3 and 4, applying the proposed approach to an existing journaling file system (either full data journaling or metadata journaling) mainly involves replacing the original code of handling journal area with invocations of the proposed journaling interface (i.e. the *VM_journal()* hypercall). Specifically, the following modifications are required. First, code for managing journal area during file system mount/unmount needs to be replaced with the invocations to the *LOG_OP_INIT* and *LOG_OP_CLEANUP* operations. Second, code of committing journal data (i.e. metadata and/or data updates) to the disk storage needs to be replaced with the invocations to the *LOG_OP_MPROTECT* operation to protect the journal data in batch and to unprotect the ODP buffers. Third, when a buffer is going to be reclaimed by the (guest) operating system, the function *reclaim_journal_data()* shown in Figure 3 needs to be invoked, which contains the invocation to the *LOG_OP_MPROTECT* operation of the hypercall for batch unprotection. Note that a journaling file system usually registers a callback function to be invoked when a buffer is going to be reclaimed so as to clean up the corresponding journaling-related information associated with the buffer. The function *reclaim_journal_data()* can simply be invoked in the callback function. Fourth, as mentioned in Section 4.2, COW is needed to handle writes to the protected buffers.

From the above description, applying the proposed approach to an existing journaling file system mainly involves replacing the original code of handling journal area with invocations of the above operations of the proposed journaling interface, which is also applicable to file systems with metadata journaling.

7. RELATED WORK

In this section, we briefly describe the related work, which can be divided into three categories: file system consistency and data integrity, NVRAM-based file and storage systems and VMA file systems.

7.1. File system consistency and data integrity

File system consistency and data integrity are critical issues for file system design. Although file system consistency can be ensured by ordered and synchronous writes, most file systems do not adopt this approach for performance consideration. On the other hand, a traditional file system that uses asynchronous writes (e.g. FFS with O_ASYNC option and ext2) needs to perform a whole file system check (*fsck*) after a system crash to bring the file system back to a consistent state, which requires a long time for file systems that manage large disk volumes.

Several approaches have been proposed to ensure file system consistency, such as journaling file systems [41], soft update [42], WAFL(Write Anywhere File Layout)-like file systems [43] and log-structured file systems [44]. Below, we briefly introduce these approaches except for the journaling file systems, which have been described in Section 2.

Soft update tracks metadata dependency to guarantee in-order update to the storage. Instead of achieving the guarantee by synchronous writes, it uses asynchronous writes and removes the dependency to the in-memory blocks when a block is going to be flushed. Specifically, in the to-be-flushed block, all the pointers that refer to the in-memory blocks are rolled back to their prior states in order to maintain consistency. The pointers are then rolled forward when the block has

DOI: 10.1002/spe

been completely written to the storage. For example, assume that a directory block is originally in state S and a file creation operation changes the directory block to state S' by associating the inode of the newly created file to the directory block. If the directory block needs to be flushed before the new inode, its state is rolled back from S' to S to avoid inconsistency. The state can be rolled forward to S' after the flush is completed. Such roll back/forward operations could lead to additional flushes for the same block. Moreover, maintaining dependency information requires extra memory and CPU resources.

WAFL-like file systems guarantee file system consistency by periodically taking file system snapshots (i.e. consistent file system states). They adopt the non-in-place update approach on storage blocks, that is, a block update is written to a new location on the storage and then reflected recursively up to the root of the file system tree. In a WAFL-like file system, a snapshot can be taken simply by duplicating the root inode. After the duplication, the old root inode corresponds to the snapshot while the new one corresponds to the current file system whose state can be changed by further file system operations. After the system crash, the latest snapshot can be used to bring the file system back to the consistent state.

The log-structured file system (LFS) regards the storage as a log and appends all the file system updates to the end of the log. The space occupied by the stale data (i.e. the garbage) is reclaimed later by a cleaner process. LFS keeps vital metadata information (e.g. inode map and segment usage table) in memory and writes the information to the log during the periodic checkpointing. In case of a system crash, the in-memory metadata can be recovered according to the last checkpoint and the following log information. Since file system updates are realized by sequential writes, LFS has excellent write performance. The main overhead of LFS is the cleaning process, which maintains enough free space in the log. Although some approaches have been proposed to reduce the cleaning overhead [45, 46], their effectiveness depends on the workload [47].

Note that not all of these approaches guarantee data integrity, which requires atomic data and metadata update of a file operation. Specifically, traditional asynchronous-write based file systems, soft update and metadata journaling cannot ensure data integrity since they do not track data updates. In contrast, full data journaling, WAFL-like file systems and LFS record new data and metadata updates before applying the updates and thus ensure data integrity.

VMA journaling aims at improving the performance of journaling file systems in virtualization environment by eliminating their main overhead (i.e. journal writes). VMA journaling supports full data journaling and hence guarantees both file system consistency and data integrity. Moreover, the proposed journaling interface allows VMA journaling to be applied to existing journaling file systems without much effort.

7.2. File/storage systems based on non-volatile memory

To prevent loss of journal data, VMA journaling retains the memory of a guest domain during the crash of that domain. This is similar to systems that store critical information in non-volatile memory to prevent the information from being lost. In this section, we describe the efforts that utilize non-volatile memory to improve the reliability or performance of file or storage systems. Previous studies such as Phoenix [48] and RIO use memory protection on battery powered memory in order to improve file system reliability. In Phoenix, two time-stamped versions of file systems, write-protected and writable versions, are maintained in memory. The former is consistent and evolves to the latter via COW. Periodic checkpointing is used to ensure file system consistency. Specifically, the writable version becomes protected and consistent after each checkpoint. In contrast to Phoenix, which is an in-memory file system, VMA journaling is a journal approach that improves the performance of on-storage journaling file systems in virtualization environments.

RIO aims at making memory as safe as disk. By maintaining a file system cache in a battery powered memory, RIO eliminates periodic dirty data flushes and sync-induced writes, thereby improving the file system performance. VMA journaling differs from RIO in two aspects. First, RIO employs *synchronous* page-level write protection and checksum to prevent unauthorized memory modifications, whereas VMA journaling protects buffer pages *asynchronously* at each journal commit. Such a difference leads to better performance with the cost of losing some data updates

DOI: 10.1002/spe

in VMA journaling. Specifically, RIO requires each authorized page update to be preceded by page unprotection and followed by page protection. According to previous studies [49, 50], such frequent page unprotection/protection degrades the file system performance dramatically in I/O-intensive workloads. With asynchronous page protection, VMA journaling reduces the performance degradation with the cost of losing the completed but not yet committed file operations upon domain crashes. Since traditional journaling file systems also lose such data, the cost is considered as acceptable with the evidence of wide application of journaling file systems. Second, RIO cannot ensure data integrity since it might restore the buffers belonging to incomplete file operations. RIO restores all the buffers without considering whether a buffer belongs to a complete file operation, which could thus lead to file data corruption. Corrupting curial configuration files such as */etc/fstab* might cause the crash of the rebooted operating system. Although a user-level library called RIO Vista [51] has been proposed to provide atomic update to memory mapped file on RIO memory, it is suitable only for applications that use memory mapped files, and it requires the applications to be modified to use the Vista interface. Moreover, additional writes to an on-storage redo log are still required. In contrast, VMA journaling provides atomic file operation that guarantees data integrity to existing applications while eliminating the additional log writes.

In addition to the above differences, VMA journaling achieves less IO traffic during the recovery when compared with RIO-based systems. During a system crash, RIO dumps *all* the memory content to the swap storage to save the protected buffers. After the reboot, the saved data are analysed to bring the buffers back to the memory. By contrast, VMA journaling only writes the dirty buffers belonging to committed file operations to the storage. As shown in Table III, in a domain with 128 MB of memory, VMA journaling flushes about 70 MB of memory in the recovery procedure even in a heavily loaded condition, a 45% reduction on the amount of I/O traffic when compared with RIO, which flushes all the domain memory (i.e. 128 MB) to the swap storage.

In addition to file system reliability, non-volatile memory has also been used to improve the file system performance. In addition to RIO, which is described above, Baker *et al.* [52] also take advantage of non-volatile memory as file caches to improve the I/O performance. The eNVy [53] storage system uses battery-backed SRAM to improve the performance of flash storage. HeRMES [54], Conquest [40], MRAMFS [55] and Kim *et al.* [56] maintain file system metadata in non-volatile RAM to reduce random accesses to the disks.

7.3. VMA file systems

In the recent years, several VMA file systems, such as VMFS [57], XenFS [58] and Ventana [59], have been proposed. All of them focus on the sharing and management of virtual disks. VMFS is a cluster file system optimized for accesses to large-size virtual disks. It allows a set of storages to be shared by multiple virtual machines via iSCSI or fibre channel links. Ventana is a distributed file system that provides fine-grained versioning and secured sharing on a set of file trees for virtual machines. XenFS allows multiple virtual machines to share the cache of virtual disks so as to reduce the accesses to the virtual disks. COW is used when a virtual machine issues a write to the shared copy. In contrast to those file systems that mainly focus on virtual disk sharing, VMA journaling aims at improving the performance of journaling file systems in a virtual machine.

8. CONCLUSIONS

In this paper, a virtual machine aware journaling approach called VMA journaling is proposed to eliminate journal writes and hence to improve the performance of journaling file systems in server virtualization environments. Based on reliable hardware subsystems and VMM, VMA journaling ensures file system consistency as well as data integrity upon virtual machine crashes. The journal writes are eliminated by placing the journal area in the VMM memory rather than on the storage. Moreover, in contrast to traditional journaling approaches that write the journal data to the journal area, only the metadata (i.e. the journal information) is written to the journal area in VMA journaling. This leads to the very small size of the journal area even in the case of full data

DOI: 10.1002/spe

journaling, and in turn causes the checkpoint procedure to be rarely triggered in VMA journaling. Page protection is used to prevent wild writes in the guest domains from corrupting the journal data, and batch protection/unprotection is adopted to minimize the protection overhead. Upon a system crash, a recovery thread locates the journal data with the support of the VMM and writes the data back to the data storage so as to recover the file system state.

VMA journaling is implemented in Linux ext3 running on the Xen VMM and the performance is evaluated via three micro- and three macro-benchmarks. Performance results show that VMA journaling achieves a performance improvement of up to 50.9% over the full data journaling approach of ext3. Under metadata-write dominated workloads, its performance could be superior even to the metadata journaling approaches of ext3, which do not guarantee data integrity. Moreover, the recovery time is less than the full data journaling approach of ext3 since VMA journaling does not need to scan the on-storage journal area for replaying the committed metadata and data updates. Finally, the memory overhead is not significant.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and the editor for their helpful comments on this paper. This research was supported in part by grant NSC 97-2221-E-006-138-MY3 from the National Science Council, Taiwan, Republic of China.

REFERENCES

1. Gray J, Reuter A. *Transaction Processing*: *Concepts and Techniques*. Morgan Kaufmann: Los Altos, CA, U.S.A., 1992.
2. Ts'o TY, Tweedie S. Planned extensions to the Linux Ext2/Ext3 filesystem. *Proceedings of the USENIX Annual Technical Conference*: *FREENIX Track*. USENIX Association: Anaheim, CA, U.S.A., 2002; 235–243.
3. Bonwick J, Moore B. ZFS: The Last Word in File Systems, 2006. Available at: http://hub.opensolaris.org/bin/download/Community+Group+zfs/docs/zfslast.pdf [June 2006].
4. Oracle Corporation. Btrfs Design, 2008. Available at: https://btrfs.wiki.kernel.org/index.php/Btrfs_design [December 2010].
5. Bryant R, Forester R, Hawkes J. File system performance and scalability in Linux 2.4.17. *Proceedings of the USENIX Annual Technical Conference*: *FREENIX Track*. USENIX Association: Anaheim, CA, U.S.A., 2002; 259–274.
6. Prabhakaran V, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Analysis and evolution of journaling file systems. *Proceedings of the USENIX Annual Technical Conference*. USENIX Association: Anaheim, CA, U.S.A., 2005; 8.
7. Bateman K. IDC charts rise of virtual machines. Available at: http://www.channelweb.co.uk/crn/news/2242378/virtual-machines-exceeding [May 2009].
8. Chen PM, Noble BD. When virtual is better than real. *HotOS '01*: *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. IEEE Computer Society: Silver Spring, MD, U.S.A., 2001; 133.
9. Swift MM, Levy HM, Bershad BN. Improving the reliability of commodity operating systems. *SOSP '03*: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM: Bolton Landing, NY, U.S.A., 2003; 207–222. DOI: http://doi.acm.org/10.1145/945445.945466.
10. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *SOSP '03*: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM: Bolton Landing, NY, U.S.A., 2003; 164–177. DOI: http://doi.acm.org/10.1145/945445.945462.
11. Oglesby R, Herold S. *VMware ESX Server*: *Advanced Technical Design Guide* (*Advanced Technical Design Guide Series*). The Brian Madden Company: Newton City, MA, U.S.A., 2005.
12. Prabhakaran V, Bairavasundaram LN, Agrawal N, Gunawi HS, Arpaci-Dusseau AC, Arpaci-Dusseau RH. IRON file systems. *SOSP '05*: *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. ACM: Brighton, U.K., 2005; 206–220. DOI: http://doi.acm.org/10.1145/1095810.1095830.
13. Intel Corporation. RAS Technologies for the Enterprise. Available at: http://www3.intel.com/Assets/PDF/whitepaper/ras.pdf [August 2005].
14. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A System Programming Guide. Available at: http://www.intel.com/assets/PDF/manual/253668.pdf [March 2010].
15. Intel Corporation. Intel E7500 Chipset MCH Single Device Data Correction (x4 SDDC) Implementation and Validation. Available at: http://www.intel.com/assets/pdf/appnote/292274.pdf [August 2002].
16. Cisco Systems Inc. Data Sheet of Cisco UCS C250 Server, 2010. Available at: http://www.cisco.com/en/US/prod/collateral/ps10265/ps10493/data_sheet_c78-559210.pdf [October 2010].
17. Schroeder B, Pinheiro E, Weber WD. DRAM errors in the wild: A large-scale fieldstudy. *SIGMETRICS '09*: *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*. ACM: Seattle, WA, U.S.A., 15–19 June 2009; DOI: http://doi.acm.org/10.1145/1555349.1555372.

18. Yunhong J, Liping K, Jinsong L. Towards mission critical Xen—RAS enabling update. *Xen Summit*. Xen, Shanghai, China, 2009. Available at: http://www.xen.org/files/xensummit_intel09/xen-summit-2009-shanghai-RAS.pdf [November 2009].
19. Heiser G, Leslie B. The OKL4 microvisor: Convergence point of microkernels and hypervisors. *APSys '10*: *Proceedings of the First Asia-Pacific Workshop on Systems*. ACM: New Delhi, India, 30 August 2010; DOI: http://doi.acm.org/10.1145/1851276.1851282.
20. Heiser G, Uhlig V, LeVasseur J. Are virtual-machine monitors microkernels done right? *ACM SIGOPS Operating System Review* 2005; **40**(1):95–99. DOI: http://doi.acm.org/10.1145/1113361.1113363.
21. Microsoft Corporation. Hyper-V Server, 2009. Available at: http://www.microsoft.com/hyper-v-server/en/us/default.aspx [September 2009].
22. Peter M, Schild H, Lackorzynski A, Warg A. Virtual machines jailed: Virtualization in systems with small trusted computing bases. *Proceedings of the First EuroSys Workshop on Virtualization Technology for Dependable Systems*. ACM: Nuremberg, Germany, 2009; 18–23. DOI: http://doi.acm.org/10.1145/1518684.1518688.
23. Steinberg U, Kauer B. NOVA: A microhypervisor-based secure virtualization architecture. *EuroSys '10*: *Proceedings of Fifth European Conference on Computer Systems*. ACM: New York, NY, U.S.A., 2010; 209–222. DOI: http://doi.acm.org/10.1145/1755913.1755935.
24. Chou A, Yang J, Chelf B, Hallem S, Engler D. An empirical study of operating systems errors. *SOSP '01*: *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. ACM: Banff, AB, Canada, 2001; 73–88.
25. Mccune JM, Li Y, Qu N, Zhou Z, Datta A, Gligor V, Perrig A. TrustVisor: Efficient TCB reduction and attestation. *SP '10*: *Proceedings of the 31st IEEE Symposium on Security and Privacy*. IEEE Computer Society: Washington, DC, U.S.A., 2010; 143–158. DOI: http://dx.doi.org/10.1109/SP.2010.17.
26. Tessin MV. Towards high-assurance multiprocessor virtualisation. *VERIFY '10*: *Proceedings of the Sixth International Verification Workshop*, Floc, Ediburgh, U.K., 2010.
27. AMD Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Available at: http://support.amd.com/us/Processor_TechDocs/24593.pdf [June 2010].
28. Intel Corporation. Intel virtualization technology. *Intel Technology Journal* 2006; **10**(3):167–177.
29. Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. SeL4: Formal verification of an OS kernel. *SOSP '09*: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM: Big Sky, Mt, U.S.A., 2009; 207–220. DOI: http://doi.acm.org/10.1145/1629575.1629596.
30. Heiser G, Andronick J, Elphinstone K, Klein G, Kuz I, Ryzhyk L. The road to trustworthy systems. *STC '10*: *Proceedings of the Fifth Workshop on Scalable Trusted Computing*. Chicago, U.S.A., 2010.
31. Brown E. Hypervisor Technology Claimed 100 Percent Bug-free. Available at: http://www.linuxfordevices.com/c/a/News/NICTA-sel4-OK-Labs-OKL4/ [13 August 2009].
32. Garfinkel T, Pfaff B, Chow J, Rosenblum M, Boneh D. Terra: A virtual machine-based platform for trusted computing. *SOSP '03*: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM: Bolton Landing, NY, U.S.A., 2003; 193–206. DOI: http://doi.acm.org/10.1145/945445.945464.
33. Chen PM, Ng WT, Chandra S, Aycock C, Rajamani G, Lowell D. The Rio filecache: Surviving operating system crashes. *ASPLOS '96*: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM: Cambridge, MA, U.S.A., 1996; 74–83. DOI: http://doi.acm.org/10.1145/237090.237154.
34. Kustarz E, Shepler S, Wilson A. Filebench: The New and Improved File System Benchmarking Framework, 2008. Available at: http://www.usenix.org/events/fast08/wips_posters/slides/wilson.pdf [February 2008].
35. Katcher J. PostMark: A new file system benchmark, 1997. Available at: http://communities-staging.netapp.com/servlet/JiveServlet/download/2609-1551/Katcher97-postmark-netapp-tr3022.pdf [August 1997].
36. Agrawal N, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Generating realistic impressions for file system benchmarking. *FAST '09*: *Proceedings of the Seventh Conference on File and Storage Technologies*. USENIX Association: San Francisco, CA, U.S.A., 2009; 125–138.
37. Pinheiro E, Weber WD, Barroso LA. Failure trends in a large disk drive population. *FAST '07*: *Proceedings of the Fifth USENIX Conference on File and Storage Technologies*. USENIX Association: San Jose, CA, U.S.A., 2007; 2.
38. Hsu ST, Chang RC. An implementation of using remote memory to checkpoint processes. *Software*: *Practice and Experience* 1999; **29**(11):985–1004.
39. Lu C. Scalable diskless checkpointing for large parallel systems. *Technical Report*, University of Illinois at Urbana-Champaign, 2005; 1–165. Available at: http://www.ideals.illinois.edu/handle/2142/11054 [August 2005].
40. Wang AA, Kuenning G, Reiher P, Popek G. The conquest file system: Better performance through a disk/persistent-RAM hybrid design. *ACM Transactions on Storage* 2006; **2**(3):309–348. DOI: http://doi.acm.org/10.1145/1168910.1168914.
41. Hagmann R. Reimplementing the cedar file system using logging and group commit. *SOSP '87*: *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. ACM: Austin, TX, U.S.A., 1987; 155–162. DOI: http://doi.acm.org/10.1145/41457.37518.
42. Ganger GR, McKusick MK, Soules CAN, Patt YN. Soft updates: A solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems* 2000; **18**(2):127–153. DOI: http://doi.acm.org/10.1145/350853.350863.

43. Hitz D, Lau J, Malcolm M. File system design for an NFS file server appliance. *Proceedings of the USENIX Winter Technical Conference*. USENIX Association: San Francisco, CA, U.S.A., 1994; 19.
44. Rosenblum M, Ousterhout JK. The design and implementation of a log-structured file system. *SOSP '91*: *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. ACM: Pacific Grove, CA, U.S.A., 1991; 1–15. DOI: http://doi.acm.org/10.1145/121132.121137.
45. Blackwell T, Harris J, Seltzer M. Heuristic cleaning algorithms in log-structured file systems. *Proceedings of the USENIX Technical Conference*. USENIX Association: New Orleans, LA, U.S.A., 1995; 23.
46. Wang J, Hu Y. WOLF—A novel reordering write buffer to boost theperformance of log-structured file system. *FAST '02*: *Proceedings of the First USENIX Conference on File and Storage Technologies*. USENIX Association: Monterey, CA, U.S.A., 2002; 4.
47. Zhang Z, Ghose K. yFS: A journaling file system design for handling large data sets with reduced seeking. *FAST '03*: *Proceedings of the Second USENIX Conference on File and Storage Technologies*. USENIX Association: San Francisco, CA, U.S.A., 2003; 59–72.
48. Gait J. Phoenix: A safe in-memory file system. *ACM Communication* 1990; **33**(1):81–86. DOI: http://doi.acm.org/10.1145/76372.76378.
49. Greenan KM, Miller EL. PRIMS: Making NVRAM suitable for extremely reliable storage. *HotDep '07*: *Proceedings of the Third Workshop on Hot Topics in System Dependability*. USENIX Association: Edinburgh, U.K., 2007; 10.
50. Wright CP, Spillane R, Sivathanu G, Zadok E. Extending ACID semantics to the file system. *ACM Transactions on Storage* 2007; **3**(2):4. DOI: http://doi.acm.org/10.1145/1242520.1242521.
51. Lowell DE, Chen PM. Free transactions with Rio Vista. *SOSP '97*: *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. ACM: Saint Malo, France, 1997; 92–101. DOI: http://doi.acm.org/10.1145/268998.266665.
52. Baker M, Asami S, Deprit E, Ouseterhout J, Seltzer M. Non-volatile memory for fast, reliable file systems. *ASPLOS '92*: *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM: Boston, MA, U.S.A., 1992; 10–22. DOI: http://doi.acm.org/10.1145/143365.143380.
53. Wu M, Zwaenepoel W. eNVy: A non-volatile, main memory storage system. *ASPLOS '94*: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM: San Jose, CA, U.S.A., 1994; 86–97. DOI: http://doi.acm.org/10.1145/195473.195506.
54. Miller EL, Brandt SA, Long DDE. HeRMES: High-performance reliable MRAM-enabled storage. *HotOS '01*: *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. IEEE Computer Society: Silver Spring, MD, U.S.A., 2001; 95.
55. Edel NK, Tuteja D, Miller EL, Brandt SA. MRAMFS: A compressing file system for non-volatile ram. *MASCOTS '04*: *Proceedings of the 12th Annual International Symposium on Modeling*, *Analysis*, *and Simulation of Computer and Telecommunications Systems*. IEEE Computer Society: Silver Spring, MD, U.S.A., 2004; 596–603.
56. Kim JK, Lee HG, Choi S, Bahng KI. A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. *EMSOFT '08*: *Proceedings of the Eighth ACM International Conference on Embedded Software*. ACM: Atlanta, GA, U.S.A., 2008; 31–40. DOI: http://doi.acm.org/10.1145/1450058.1450064.
57. Clements AT, Ahmad I, Vilayannur M, Li J. Decentralized deduplication in SAN cluster file systems. *Proceedings of the USENIX Annual Conference*, San Diego, CA, U.S.A., 2009; 101–114.
58. Williamson M. XenFS, 2008. Available at: http://wiki.xensource.com/xenwiki/XenFS [August 2007].
59. Pfaff B, Garfinkel T, Rosenblum M. Virtualization aware file systems: Getting beyond the limitations of virtual disks. *NSDI '06*: *Proceedings of the Third Conference on Networked Systems Design and Implementation*. USENIX Association: San Jose, CA, U.S.A., 2006; 26.

DOI: 10.1002/spe