Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment

Stephen T. Jones   Andrea C. Arpaci-Dusseau   Remzi H. Arpaci-Dusseau

Computer Sciences Department, University of Wisconsin–Madison

{stjones, dusseau, remzi}@cs.wisc.edu

Abstract

Virtualization is increasingly being used to address server management and administration issues like flexible resource allocation, service isolation and workload migration. In a virtualized environment, the virtual machine monitor (VMM) is the primary resource manager and is an attractive target for implementing system features like scheduling, caching, and monitoring. However, the lack of runtime information within the VMM about guest operating systems, sometimes called the semantic gap, is a significant obstacle to efficiently implementing some kinds of services.

In this paper we explore techniques that can be used by a VMM to passively infer useful information about a guest operating system's unified buffer cache and virtual memory system. We have created a prototype implementation of these techniques inside the Xen VMM called Geiger and show that it can accurately infer when pages are inserted into and evicted from a system's buffer cache. We explore several nuances involved in passively implementing eviction detection that have not previously been addressed, such as the importance of tracking disk block liveness, the effect of file system journaling, and the importance of accounting for the unified caches found in modern operating systems.

Using case studies we show that the information provided by Geiger enables a VMM to implement useful VMM-level services. We implement a novel working set size estimator which allows the VMM to make more informed memory allocation decisions. We also show that a VMM can be used to drastically improve the hit rate in remote storage caches by using eviction-based cache placement without modifying the application or operating system storage interface. Both case studies hint at a future where inference techniques enable a broad new class of VMM-level functionality.

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

General Terms Design, Measurement, Performance

Keywords Virtual Machine, Inference

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

Reprinted from ASPLOS'06, Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, October 21–25, 2006, San Jose, California, USA., pp. 13–23.

1. Introduction

Virtualization technology is an increasingly common component in server and desktop PC systems. As both software [13, 39] and hardware [18, 20] support for low-overhead virtualization develops, and as virtualization is included in popular commercial systems [3], we expect virtualized computing environments to become ubiquitous.

As virtualization becomes prevalent, the *virtual machine monitor* (VMM) naturally supplants the operating system as the primary resource manager for the machine. Hence, the VMM becomes an attractive target for implementing what would traditionally be considered operating system features like flexible resource management [38], service and device driver isolation [15], load balancing [10], security monitoring [16, 24], and fault tolerance [5].

The transition of some functionality from the OS into the VMM has many potential benefits. For example, because a guest operating system is isolated from the VMM by the stable virtual hardware interface, services implemented within a VMM can be portable across many guest operating systems. Further, the VMM may be the only place where innovative features can be inserted into a system, because the guest operating system is legacy or closed-source or both. Finally, in a virtualized environment, the VMM is the only component that has total control over system resources and can therefore likely make the most informed resource management decisions.

Pushing functionality down one layer in the software stack into the VMM has disadvantages as well. One significant problem is the lack of higher-level knowledge within the VMM, usually referred to as a *semantic gap* [7]. Previous work in virtualized environments has partially recognized this dilemma, and researchers have developed techniques to infer information about how guests utilize virtual hardware resources [6, 30, 38]. These techniques are useful because they allow a VMM to manage the resources of a system more effectively (*e.g.*, by reallocating an otherwise idle page in one virtual machine to a different virtual machine that could use it [38]).

This paper describes techniques that can be used by a VMM to infer [1] information about one performance-critical *software* component, the operating system buffer cache and virtual memory system. Specifically, we show how a VMM can carefully observe guest operating system interactions with virtual hardware like the MMU and storage devices and detect when pages are inserted into or evicted from the operating system buffer cache.

Geiger is a prototype implementation of these techniques within the Xen virtual machine monitor [13]. In this paper, we discuss the details of Geiger's implementation and perform a careful evaluation of Geiger's eviction detection techniques. A few of Geiger's inferencing techniques within the VMM are similar to those used by Chen *et al.* within a pseudo-device driver [9]. Hence, our evaluation focuses on which of Geiger's new techniques are needed in different circumstances. First, we show that the unified buffer caches and virtual memory systems found in modern operating systems require the VMM to track not only disk traffic, but memory allocations as well. Second, we show that a VMM must take basic storage system behavior into account to accurately detect cache eviction. For example, the VMM must track whether a particular data block is live or dead on disk in order to avoid reporting many spurious evictions. We also show that journaling file systems, such as ext3 in Linux, require the VMM to distinguish between writes to the journal and writes to other parts of storage to avoid an aliasing problem that leads to false eviction reporting. In summary, passively detecting cache events within modern operating systems requires new sophistication. Without these techniques, passive inferencing can result in inaccurate information which can be worse than no information at all.

Via case studies, we demonstrate how the inferred eviction information provided by Geiger can enable useful services inside a VMM. In the first case study we implement a novel, VMM-based working set size estimator that complements existing techniques [38] by allowing estimation in the case that a virtual machine is thrashing. A second study explores how Geiger-inferred evictions can be used by a VMM to enable remote storage caches to implement eviction-based cache placement [40] without changing the application or operating system storage interface, hence enhancing the adoption of this feature.

The rest of this paper is organized as follows. We begin by presenting some extended motivation for why eviction information is useful and address related work in Section 2. In Section 3, we describe the techniques used by Geiger in detail. Section 4 discusses the implementation of Geiger and in Section 5 we evaluate the accuracy and overhead of our prototype Geiger implementation. Sections 6 and 7 discuss our two case studies. Finally, in Section 8 we summarize and conclude.

2. Motivation and Related Work

It is useful for a VMM to understand how its hosted virtual machines use memory. In this section, we describe two different contexts where a VMM can exploit information about buffer cache promotion and eviction events.

2.1 Working Set Size

In a virtualized environment, knowing the *working set size* [11, 12] of a virtual machine is useful for allocating the appropriate amount of memory to it. As hosted environments become more popular, it will become more and more common for many VMs to be running simultaneously on the same physical host. In this situation, knowing the working set size of each VM allows the VMM to allocate, or dynamically re-allocate, an appropriate amount of memory to each competing VM. When migrating VMs [10, 29], for example in a grid computing environment [14, 41], working set size information enables the job scheduler to intelligently select a new host with an adequate amount of available memory.

Techniques for estimating the working set size of a virtual machine have been explored by Waldspurger and are part of the VMware ESX Server [38] product. However, the ESX server technique can only determine the working set size for virtual machines that are using less than their full allocation. If a guest begins to thrash, the sampling technique used by ESX server simply reports a working set that is 100% of the VM's memory allocation, when in fact it is larger. Therefore, to determine the true working set size, the ESX technique would need to use trial and error: the VMM would repeatedly give a thrashing virtual machine incrementally more memory and re-measure the working set size until it drops below 100%. If the system does not contain enough physical memory to accommodate the full working set, then even this trial and error method will fail.

In contrast, if the VMM can detect the eviction events of each virtual machine, the VMM can directly model the hit rate of each virtual machine as a function of the amount of memory. As a result, the VMM can quickly and efficiently determine how much physical memory to give each competing virtual machine or where to migrate a virtual machine.

2.2 Secondary-level Caching

In a virtualized environment, knowing the contents of the OS buffer cache is useful when implementing an effective secondary-level cache. For example, when multiple VMs run on the same machine, the VMM can manage a shared secondary cache in its own memory, increasing the utilization of memory when the VMs share pages [6]. Additionally, when the hosted OS is a legacy system that cannot address a large amount of memory, a secondary cache can enable the legacy OS to exceed its natural addressing limits. Finally, the VMM can explicitly communicate with a remote storage server cache, informing it of which pages are currently cached within each VM [40].

Designing a secondary cache management policy is non-trivial. Secondary storage caches exhibit less reference locality than client caches because the reference stream is filtered through the client cache [25]. This, plus the fact that secondary storage caches are often about the same size as client caches has led to innovations in cache replacement policies [42] and in cache placement policies [40]. One promising placement policy, called "eviction-based placement", inserts blocks into the secondary cache only when they have been evicted from the client cache. This approach tends to make the caches overlap less and leads to more effective secondary cache utilization [9, 40]. Eviction-based placement is similar to micro-architectural victim caches in the processor cache hierarchy [23].

If one is willing to modify the OS, then eviction-based cache placement is relatively straight-forward. Wong *et al.* [40] extend the block-based storage interface with a DEMOTE operation that explicitly notifies interested parties, like remote storage caches, when pages are evicted from client caches. Our goal is not to modify the OS, hence, the VMM instead uses passive techniques to infer that a page has been evicted from the OS page cache; the VMM itself can then explicitly notify the storage cache.

Passive eviction detection has been explored to some extent for exclusive caching in storage systems. For example, X-RAY [2] uses file system semantic information (*e.g.*, which storage blocks contain inodes) to snoop on updates to a file's accessed time field. Knowing which files have been recently accessed allows X-RAY to build an approximate model of a client's cache. However, X-RAY is somewhat limited in its inferences because the storage system only has access to the I/O block stream outside the OS.

Other exclusive caching work has assumed that one has access to more OS information; for example, Chen *et al.* [8, 9] perform their inferencing within a pseudo-device driver that has access to the addresses of the memory pages that are being read and written. Thus, they are able to infer that an eviction has occurred when a memory page that is storing disk data is reused for other disk data.

The Geiger approach is most similar to that of Chen *et al.*, but uses additional information available to a VMM to improve its ability to infer cache events. Geiger builds upon previous work in three important ways. First, Geiger handles guest operating systems that implement a unified buffer cache and virtual memory system. With a unified buffer cache, events other than disk I/O can evict pages; thus, Geiger contains new techniques to handle when anonymous memory allocation causes cache evictions. Second, Geiger recognizes when blocks on disk are freed. If a block is free on disk, reuse of a memory buffer that recently held its contents does not imply eviction; this distinction is important as it is counterproductive to cache blocks the file system believes are free. Third, Geiger supports journaling file systems. When the file system writes the same block to two distinct locations on disk, as occurs with a journaling file system, Geiger avoids reporting false evictions. Thus, with these three additional techniques, Geiger is able to handle the range of situations that occur in modern operating systems.

3. Geiger Techniques

In this section we discuss the techniques used by Geiger. We begin by providing relevant background about virtual machine monitors, and continue by describing the basic techniques Geiger uses to infer page cache promotion and eviction. We then describe how Geiger performs more complex inferences, in particular, how it handles unified buffer caches and virtual memory systems that are present in all modern operating systems, and how it handles issues that arise due to storage system interactions.

3.1 Background

Virtualization is a well studied technique [17] that has somewhat recently been revitalized in the ubiquitous PC environment [6, 33, 38, 39]. In a virtualized environment, a thin layer of software known as the *virtual machine monitor* (VMM) virtualizes some components of a host computer system allowing one or more *guest* operating systems to safely share available resources transparently.

A key feature of a virtualized environment is that guest operating systems execute using the unprivileged mode of the processor while the VMM runs with full privilege. Hence, guest OS accesses to sensitive system components (*e.g.*, the MMU and I/O peripherals) cause an exception and control is transferred to the VMM. At this point, the VMM can decide the appropriate action (*e.g.*, emulate the behavior of the privileged hardware or operation in software).

In this paper, we take advantage of these VMM entry points to observe interesting OS activity. For example, Geiger observes architecturally visible events such as page faults, page table updates, disk reads, and disk writes to infer the occurrence of a buffer cache eviction or promotion. One major benefit of observing only these entry points is performance; adding a small piece of observation code at these points induces little or no overhead.

3.2 Basic Techniques

Buffer cache *promotion* occurs when a disk page is added to the cache. Buffer cache *eviction* occurs when a cache page is freed by the operating system and its previous contents remain available to be reloaded from disk. For example, an eviction occurs if the contents of an anonymous page are written to a swap partition and then the page is freed. Similarly, an eviction occurs if a page that was read from the file system is later freed without writing anything back to disk, since the data can be reloaded from the original location on disk. However, an eviction does not occur if the OS frees a page and its contents are lost (*e.g.*, an anonymous page when its associated process exits).

To detect promotion and eviction, Geiger performs two tasks. First, Geiger tracks whether the contents of a page are available on disk and, if so, where on disk the contents are stored. We call the on-disk location associated with a memory page the page's *Associated Disk Location* (ADL). Second, Geiger must detect when a page is freed by the OS. We describe each of these steps in turn.

3.2.1 Associated Disk Locations

Geiger associates a disk location with each physical memory page, whenever appropriate. An associated disk location (ADL) is simply the pair `<device, block offset>`, representing the most recent disk location with which a VMM can associate the page. A VMM associates a disk location with a memory page whenever that page is involved in a disk read or write operation. For example, if a page is the target of a read from disk location A, the page becomes associated with A. Similarly, if a page is the source of a write to disk location B the page becomes associated with B. These associations persist until replaced by another association, the memory page is freed, or the relevant disk blocks are freed.

Since the VMM virtualizes all disk I/O, disk reads and writes initiated by a guest are explicitly visible to the VMM and no special action on the part of the VMM is required to establish the ADL of a page. However, to correctly invalidate an ADL when the disk block, to which it refers, is no longer in use requires detecting when the disk block is freed. We discuss this further in Section 3.4.

3.2.2 Detecting Page Reuse

Geiger must also determine when a memory page has been freed by the OS. However, the guest OS does not explicitly notify the VMM when it frees a page. Often the only difference between an active and a free page is an entry in a private OS data structure, such as a free list or bitmap. We assume that the VMM does not have the detailed, OS-specific information required to locate or interpret these data structures. Hence, instead of detecting that a page has been freed, Geiger detects that a page has been *reused*. Since reuse implies that a page was freed between uses, it is an appropriate proxy for the page free event.

Geiger uses numerous heuristics to detect that a page has been reused. Each heuristic corresponds to a different scenario in which a guest OS allocates a page of memory. If Geiger detects a page allocation and the newly allocated page has a current ADL, then Geiger signals that the previous contents of the page, as defined by the ADL, have been evicted.

The two most basic techniques used by Geiger are monitoring disk reads and disk writes. This builds on the previous work of Chen *et al.* [9] which monitors reads and writes in a device driver within an OS.

Disk Read: Geiger uses disk reads to infer that a new page may have been allocated. When a page is read from disk, a new page is allocated in the OS buffer cache. If the allocated page has a current ADL that refers to a different disk location than the one currently being read, Geiger reports that the page's previous contents have been evicted. The ADL of the affected page is updated to point to the new disk location as a consequence of this kind of eviction.

Disk Write: Geiger uses disk writes to infer that a new page may have been allocated. If a full page of data is written to disk and the page does not already reside in the page cache, then the OS may allocate a new page to buffer the data until it is asynchronously written to disk. Geiger detects this case by observing all disk writes and signaling an eviction if the write source is a page with a current ADL that is different than the target disk location of the write. Note that if a previous read or write caused the disk block to already exist in the cache, Geiger will not erroneously signal a duplicate eviction since the page's ADL will not change. As with the read-eviction heuristic, the ADL of the affected page is updated to refer to the target disk location.

3.3 Techniques for Unified Caches

Techniques from previous research [9] work well with old-style file system buffer caches, which were kept distinct from the virtual memory system. However, virtually all modern operating systems, including Linux, *BSD, Solaris, and Windows, have a unified buffer cache and virtual memory system. Unification complicates inferences: Geiger must be able to detect page reuse for additional cases associated with the virtual memory system. Hence, we introduce two new detection techniques.

Microbenchmark	Description
Read Evict	Sequentially reads a section of a file larger than available memory multiple times
Write Evict	Sequentially writes a file larger than available memory. Repeated multiple times.
COW Evict	Allocates a memory buffer approximately the size of available physical memory, then writes to each virtual page to ensure a physical page is allocated, then forks and writes to each page in the child.
Allocation Evict	Allocates a memory buffer that exceeds the size of available memory and writes to each virtual page to ensure a page is allocated.

Figure 1. Microbenchmark Workloads. *This table describes the four microbenchmarks used to isolate a specific type of page eviction.*

Application	Description	Read %	Write%	COW%	Alloc%
Dbench [35]	File system benchmark simulating load on a network file server	41.13%	58.85%	0.00%	0.00%
Mogrify [19]	Scales and converts a large bitmap image	53.22%	22.31%	0.01%	24.25%
OSDL-DBT1 [26]	TPC-W-like web commerce benchmark simulating web purchase transactions in an online store.	77.02%	2.14%	0.54%	20.29%
SPC Web Search 2 [32]	Storage performance council block device traces from a web search engine server. Traces are replayed to a real file system.	99.6 %	0.03%	0.00%	0.00%

Figure 2. Application Workloads. *This table describes each of the four application workloads and reports the percentage of total eviction events caused by each eviction type.*

Copy On Write: Copy-on-write (COW) is a technique widely used in operating systems to implement efficient read sharing of memory. A page shared using COW is marked read-only in each process's virtual address space that shares it. When one of these processes attempts to write to a COW-shared page, the action causes a page fault. The operating system then transparently allocates a new, private page and copies the data from the old page into the new page. Subsequently, a new writable virtual memory mapping is established which refers to the new page. Because the private copy requires allocation of a free page, it can lead to page reuse.

Geiger detects page reuse that occurs as a result of COW by observing page faults and page table updates. When Geiger detects a page fault whose cause is a write into a read-only page, it saves the affected virtual address and page table entry in a small queue. If, a short time later, the guest OS creates a new writable mapping for the same virtual address, but a different physical page, Geiger infers that the new physical page was newly allocated. If the newly allocated page has an active ADL, then Geiger signals an eviction. This heuristic clears the ADL of the newly allocated page because it is a modified private copy of an existing page and is not associated with any disk location.

Allocation: Most modern operating systems allocate memory lazily. When an application requests memory (*e.g.*, using `brk` or an anonymous `mmap`), the OS does not immediately allocate physical memory; instead the virtual address range is "reserved" and physical memory is allocated on-demand when the page is actually accessed. This property means that physical memory allocation nearly always occurs in the context of servicing a page fault.

Similar to the COW heuristic, Geiger observes page faults that are due to a guest accessing a virtual page that has no virtual-to-physical mapping and saves the affected virtual address in a small queue. If, a short time later, the guest OS creates a new *writable* mapping for the faulting virtual address, Geiger infers a page allocation. If the newly allocated physical page has a current ADL, then Geiger signals an eviction.

3.4 Techniques for Storage

Storage systems also introduce some nuances into the inferences made by Geiger. In particular, file system features like journaling lead to an *aliasing* problem; further, the fact that disk blocks can be deleted leads to the problem of *liveness detection*. We now describe these issues and how Geiger handles them in turn.

3.4.1 Journaling

The basic write heuristic signals an eviction whenever the contents of a page that has an ADL are written to a location on disk which does not match that ADL. For example if a page has ADL A and is written to disk location B an eviction will be reported for the contents of disk location A. The basic write heuristic over-reports evictions in cases where data are written from the same buffer cache page to multiple disk locations; we view this as an *aliasing* problem, as the same page is wrongly associated with two disk addresses.

Journaling file systems, such as Linux ext3 [36], ReiserFS [28], JFS [4], and XFS [34], routinely write to two locations on disk from the same cache page, namely the journal location and the fixed disk location. In the worst-case journaling scenario, where both data and metadata are first written to the journal, twice the actual number of write evictions will be reported. In the more common case of metadata-only journaling, a much smaller penalty is incurred.

The negative effect of journaling and virtual memory can be mitigated if the VMM identifies reads and writes to the file system journal. This is straightforward in most systems, since the journal is either placed on a separate, easily identifiable partition or in a file within a file system partition to which a reference is made from the file system superblock [37]. Hence, to avoid the problem of journal aliasing, Geiger monitors the disk addresses of write requests and ignores writes directed to the journal.

3.4.2 Block Liveness

Geiger signals that a page has been evicted only if that page has a current ADL. It is possible that the blocks to which an ADL refers are deallocated on disk between the time that the ADL mapping is first established and when Geiger detects that the associated page has been reused. In this case, Geiger will falsely report an eviction, because an ADL exists but the data to which the ADL refers have been deallocated and are no longer accessible. This problem of *block liveness* can lead to large numbers of false evictions for workloads in which files are regularly deleted, truncated, or when processes die that have significant parts of their virtual memory swapped to disk.

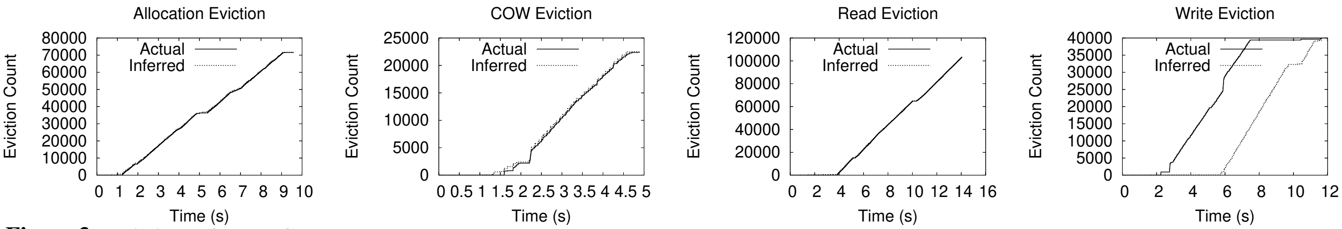

Figure 3. Eviction Inference Counts. *The figure compares inferred vs. actual eviction counts over time for microbenchmarks that isolate each eviction type inferred by Geiger.*

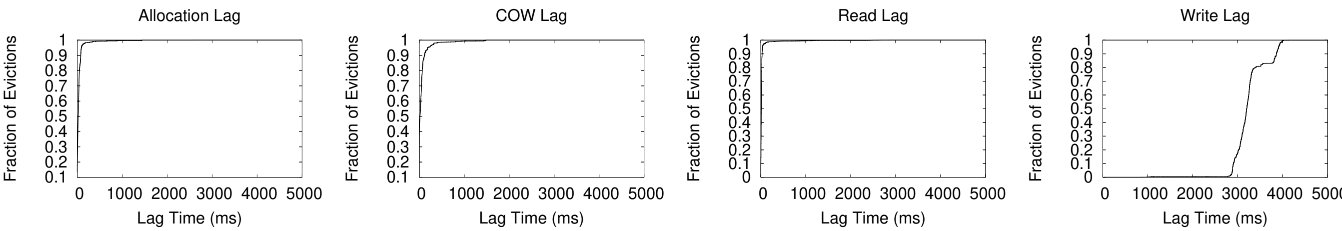

Figure 4. Eviction Lag. *The figure shows the cumulative lag distribution for microbenchmarks that isolate each eviction type.*

File systems: A virtual machine monitor can passively track *file system* block liveness in the same way a smart disk system can track block liveness [30]. The allocation state for each file system block is typically noted in some on-disk structure like a bitmap. The file system superblock, which is stored at a known, fixed location on disk can be used to locate these bitmap structures. By examining guest operating system writes to these on-disk areas, a VMM can snoop on the file system to determine when disk blocks to which an ADL refers have been freed. If the blocks to which an ADL refers are deallocated, the ADL must be invalidated so that a future reuse of the affected page is not misinterpreted as an eviction.

Implementing block liveness by observing only disk writes has one significant drawback because of the often substantial lag between when a file system structure like an allocation bitmap is updated in memory and when it is written to disk. In many operating systems this interval can be 30 seconds or more. If Geiger does not observe that the file system blocks, to which a page's ADL points, have been deallocated until after the page has been reused a false eviction will occur. Hence, the timeliness of block deallocation notification is important.

A VMM can improve the timeliness of block deallocation notification by tracking updates to the in-memory versions of the allocation bitmaps. Given the known locations of the bitmaps, the VMM can observe when bitmaps are loaded from disk into memory. At that time, the VMM can mark all such buffers read-only. When a guest updates an in-memory bitmap, a minor page fault will occur. The VMM can observe that the fault is due to an attempted bitmap update and respond by invalidating affected ADLs.

Geiger implements this style of in-memory block liveness tracking. Bitmap blocks are identified by reading and parsing the file system superblock for known file system types. Pages used to cache file system allocation bitmaps are marked read-only in memory by Geiger. When a write to such a page is detected due to a page protection fault, the effect of the faulting instruction is emulated on the guest memory and register state and the faulting instruction is skipped; hence, every bitmap update is synchronously observed and handled by the VMM. The overhead of block liveness tracking is kept low in spite of additional minor page faults due to the relatively low frequency of disk bitmap updates.

Like Sivathanu [31], we consider embedding file system layout information, such as the format of the superblock, within a VMM a reasonable technique. There are few commonly used file systems and the on-disk data structure formats for those file systems change slowly. A VMM can be provided with layout information for all commonly used file systems and the information can be expected to remain valid for a long time. The on-disk format of ext2, for example, has not changed since its introduction in 1994. This is a far longer interval than the typical system software upgrade cycle.

Swap space: The liveness tracking techniques Geiger uses for file system partitions do not apply to disk space used as a swap area. As a rule, swap space does not contain any on-disk data structures that track block allocation because data in swap is not expected to persist across system restarts. All swap allocation information is managed exclusively in volatile system memory. There are two swap liveness tracking techniques we have found to be effective for some workloads in preventing false evictions due to ADLs that point to deallocated swap space.

The first technique invalidates any ADL that points to a set of disk blocks that is overwritten. When disk blocks are overwritten, the data to which an ADL refers has been destroyed; hence, ADL invalidation is appropriate. This technique is implemented by maintaining a reverse mapping between cached disk blocks and ADLs.

The second technique makes use of implicitly obtained process lifetime information like that provided by Antfarm [22]. Given accurate information about guest OS processes and a mapping of memory pages to the owning OS process, many ADLs can be invalidated when the process exits. Specifically, an ADL from a page belonging to a dead process that points to a swap space disk block can be invalidated. This second technique appears promising but has not been fully implemented in the current version of Geiger.

4. Implementation

Geiger is implemented as an extension to the Xen virtual machine monitor version 2.0.7. Xen [13] is an open source virtual machine monitor for the Intel x86 architecture. Xen provides a paravirtualized [39] processor interface, which enables lower overhead virtualization at the expense of porting system software. We explicitly do *not* make use of this feature of Xen; hence, the mechanisms we describe are equally applicable to a more conventional virtual machine monitor such as VMWare [33, 38].

Geiger consists of a set of patches to the Xen hypervisor and Xen's block device backends. Changes are concentrated in the handlers for events like page faults, page table updates and block device reads and writes. The Geiger patches consist of approximately

Workload	False Neg %	False Pos %
Read Evict	0.96%	0.58%
Write Evict	1.68%	0.03%
COW Evict	2.47%	1.45%
Alloc Evict	0.17%	0.17%

Figure 5. Microbenchmark Heuristic Accuracy. *The table reports the false positive and false negative ratios for the complete set of eviction heuristics for each of the microbenchmark workloads.*

Workload	w/o Journal Opt		w/ Journal Opt	
	F. Neg %	F. Pos %	F. Neg %	F. Pos %
No Journal	1.68%	0.03%	1.68%	0.03%
Metadata	1.83%	0.33%	0.61%	0.08%
Data	1.43%	61.91%	2.51%	0.06%

Figure 6. Effect of Journaling. *The table reports the false positive and false negative ratios for the write-eviction microbenchmark workload when run with no journaling, with metadata journaling (ordered mode), and data journaling with the Linux ext3 file system. The table shows the benefits of turning on the Geiger specialization to detect writes to the journal.*

700 lines of code across three files. About 25 other files from the Xen hypervisor and the Linux kernel required small changes in order to implement instrumentation and tracing.

All experiments described in this paper were performed on a PC with a 2.4 GHz Pentium IV processor, 2 GB of system memory, and two WD1200BB ATA disk drives. We used Linux kernel version 2.6.11 in the Xen control domain and Linux kernel version 2.4.30 for all unprivileged domains. We use either the ext2 or ext3 file system, depending upon the experiment. The Xen control domain is configured with 512 MB of memory. Unless otherwise noted, each unprivileged guest virtual machine is assigned 128 MB of memory.

5. Evaluation

In this section we evaluate the ability of Geiger to accurately infer page cache evictions and promotions occurring within guest operating systems. We begin by describing our workloads and metrics; we then evaluate Geiger using a set of four microbenchmarks and four application workloads. We conclude by measuring the overheads that Geiger imposes on the system.

5.1 Workloads

Throughout the experimental evaluation of Geiger, we use two sets of workloads. The first workload set consists of four microbenchmarks. Each of these four microbenchmarks have been constructed to generate a specific type of page cache eviction: Read, Write, Copy-On-Write (COW), or Allocate. Thus, these microbenchmarks isolate Geiger's ability to track evictions due to specific events. The microbenchmarks are described in more detail in Figure 1.

The second set of workloads consists of four application benchmarks. These represent more realistic workloads. Each workload contains a mix of eviction types, whether read, write, COW, or allocation. Figure 2 lists the application workloads as well as the breakdown of eviction types generated by them. Thus, these application workloads stress Geiger's ability to track evictions that may occur for several different reasons.

5.2 Metrics

Our methodology for evaluating the accuracy of Geiger is to compare the trace of evictions signaled by Geiger to a trace of evictions produced by the guest operating system; we have modified the Linux kernel to generate this trace. Since the guest operating system has complete information about which pages are evicted and when, our comparison is against the ideal eviction detector.

Workload	Geiger Opts	False Neg %	False Pos %
Dbench	w/o block liveness	1.10%	30.23%
Dbench	w/ block liveness	2.30%	5.72%
Mogrify	w/o block liveness	0.05%	22.99%
Mogrify	w/ block liveness	0.65%	2.46%
TPC-W		0.14%	3.12%
SPC Web 2		2.24%	0.32%

Figure 7. Application Heuristic Accuracy. *The table reports the false positive and false negative ratios for Geiger on the four application workloads. For the Dbench and Mogrify workloads, we evaluate Geiger both without and with the optimizations to detect whether a block is live on disk.*

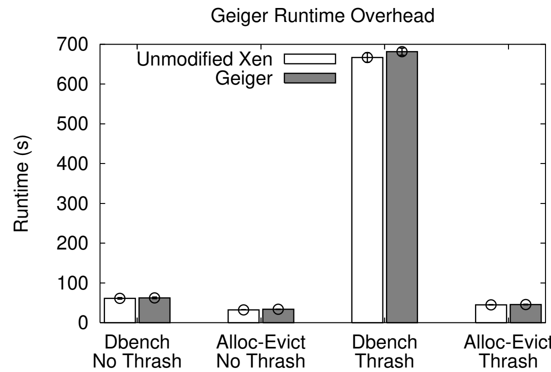

Figure 8. Geiger Runtime Overhead. *The figure shows that Geiger imposes very small runtime overheads for two workloads that stress its inference heuristics.*

The eviction records in both traces contain the physical memory address, the disk address of the evicted data, and a time stamp.

We consider three different metrics for accuracy. The first metric is simply the *eviction count* reported by Geiger compared to that reported by the guest OS over time. The second metric is *detection lag*, or the time between when a particular eviction takes place in the OS and when it is detected by Geiger. Finally, the third metric is the *detection accuracy*, which tracks the percentage of records from the inferred and actual traces that match in a one-to-one mapping; we report both the percentage of false negatives (*i.e.*, actual evictions not detected by Geiger) and false positives (*i.e.*, inferred evictions that did not correspond to OS-reported evictions).

5.3 Microbenchmarks

We begin by running workloads consisting of the four microbenchmarks. Figure 3 shows the resulting eviction count time-lines. For all microbenchmarks, the eviction counts inferred by Geiger closely match the actual OS counts; however, depending upon the workload, some interesting differences may occur along the way. For example, during the COW workload, the guest OS reclaims pages in groups, leading to a slight stair-step eviction pattern; Geiger's inferences lag slightly behind in this case. In the write workload, the guest OS begins evicting pages early and continues to evict eagerly throughout the experiment; these pages are not reused for some time. Because Geiger's inferences are based on page reuse, eviction is not detected until a page is reused, and inferred evictions lag noticeably behind actual evictions when caused by writes.

Figure 4 shows the cumulative distributions of eviction lag times for each of the microbenchmarks. As expected, the lag times for read, COW, and allocation eviction are concentrated at very small values. However, the lag times for the write microbenchmark are concentrated at about three seconds due to the operating system's eager reclamation behavior.

Figure 5 reports Geiger's detection accuracy in both false negatives and false positives. For all workloads, false negatives are uncommon: at worst, fewer than 2.5% of the total number of evictions are missed by Geiger. False positives are even less common: at worst, Geiger over-reports 1.45% of its inferred evictions.

In our final microbenchmark experiment, we explore Geiger's ability to detect aliased writes to the file system journal. We use the write workload to stress this detection. Figure 6 shows the accuracy of Geiger with and without the specialization to disregard write traffic to the file system journal. Without this specialization, Geiger performs satisfactorily when journaling is disabled or when only metadata is journaled (*i.e.*, Linux ext3 ordered-mode); with metadata journaling, relatively few blocks have aliases. However, with data journaling, many blocks have aliases and, as a result, more than half of the evictions reported by the un-specialized Geiger are false positives. In contrast, the full version of Geiger accurately handles all journaling modes of Linux ext3; even with data journaling, Geiger has a false positive percentage of only 0.06%.

5.4 Application Benchmarks

We next consider workloads containing more realistic applications. Figure 7 reports the detection accuracy of Geiger on these application workloads. For all workloads, false negative ratios are small: in the worst case, Geiger misses only 2.24% of the evictions reported by the OS. However, the Dbench and Mogrify workloads have interesting behavior regarding false positives.

5.4.1 Block Liveness

The Dbench and Mogrify workloads illustrate the benefit of having Geiger attempt to track the liveness of each block on disk. Dbench creates and deletes many files; as a result, many pages in memory are reused for different files (and different disk blocks). Mogrify causes large amounts of swap to be allocated and deallocated during its execution. If the VMM uses only the change in association between a memory page and its disk block to infer an eviction, then the VMM concludes that many evictions have occurred that actually have not (*i.e.*, many false positives). Thus, without live block detection, Geiger has a 30.2% false positive rate for Dbench and a 23% false positive rate for Mogrify. However, when Geiger tracks whether a particular disk block is free, it can detect when a page is simply reused without the previous contents being evicted; as a result, the false positive rate improves dramatically to 5.7% for Dbench and 2.46% for Mogrify. Thus, to adequately handle delete-intensive (or truncate-intensive) workloads, Geiger includes techniques to track disk block liveness.

5.4.2 Limitations

As mentioned previously, we do not expect our current techniques for tracking block liveness in swap space to be adequate in all situations. To demonstrate this remaining problem, a microbenchmark was crafted that results in large numbers of false positives despite the best efforts of Geiger to track block liveness. The program forces a large buffer (allocated using `mmap`) to be swapped to disk and then the buffer is released. In Linux, as the buffer is released, the associated swap space is also deallocated, but Geiger does not detect that event. As additional memory is allocated by the program, pages are reused whose ADLs point to deallocated swap space resulting in an eviction false positive ratio of about 37%.

5.5 Overhead

Geiger observes events that are intrinsically visible to a VMM like page faults, page table updates, and disk I/O. Except in the case of disk block liveness tracking, no additional memory protection traps or I/O requests are caused by Geiger. Liveness tracking imposes one additional minor page fault for each disk bitmap update which occur relatively rarely. Hence, we expect the runtime overhead imposed by Geiger to be small. To validate this expectation, we compare the runtime of workloads running on an unmodified version of Xen to that of Geiger. We are interested in two performance regimes. The first regime is the more common case, in which a workload has sufficient memory and few evictions occur. The second regime occurs when a machine is thrashing, since this implies that many evictions are taking place and Geiger's inference mechanisms are being stressed.

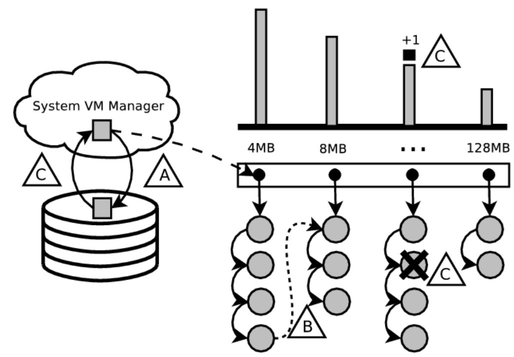

Figure 9. **MemRx Operation.** *The figure shows a schematic of the cache simulation implemented by MemRx. A) When a page is evicted by a guest, this event is detected by Geiger and an entry is added to the head of a series of queues. B) If necessary, queue entries ripple from the tail of one queue to the head of the next. C) Upon reload, the associated queue entry is removed and an array entry associated with that queue is incremented. Each entry tracks which sub-queue it appears in to enable fast depth estimation.*

We evaluate each of these four cases using two carefully chosen workloads. Since Geiger interposes on code paths for handling page faults, page table updates and disk I/O, we use the microbenchmark "allocation-evict" described in Figure 1 and Dbench described in Figure 2. Allocation-evict causes many page faults and page-table updates stressing that portion of Geiger's inference machinery. Dbench causes a large number of file creations, reads, writes, and deletes which exercise those portions of Geiger's heuristics.

Figure 8 shows the results of the experiment. Each value shown is the average of five runs; the standard deviation is shown with error bars. The largest observed overhead is 2.19%, which occurs for a thrashing Dbench. For all cases, the results for Geiger and the unmodified Xen are comparable.

Geiger requires some extra space per physical memory page to track ADLs. In our prototype this amounts to 20 bytes per memory page. In our test system, configured with 2 GB of physical memory, a total of 10 MB of additional memory is allocated by the VMM, leading to a space overhead of approximately 0.5%. If this space overhead is a concern, it could be substantially reduced, given the preallocated, fixed size, and sparsely-populated data structures of our prototype.

5.6 Hardware Trends

Major microprocessor vendors like Intel, AMD, and IBM have begun to include optional hardware virtualization features in their server and desktop products to reduce the overhead imposed by virtualization. Some of the new features can hide information from the VMM in favor of reducing guest-to-VMM transitions. For example, in some cases page faults and guest page table updates may not cause the VMM to be invoked. Geiger uses page fault and page table update information to detect cache eviction events and its absence would impact Geiger's functionality. In the future, new

Benchmark	Activity
FS Sequential	Sequentially scan a 256 MB section of a file system file 10 times
VM Sequential	Sequentially scan 256 MB section of allocated virtual memory 10 times
FS Random	Randomly read page-sized blocks from a 256 MB file system file two times
VM Random	Randomly touch virtual memory pages from 256 MB virtual memory allocation 2 times

Figure 10. Calibrated Microbenchmarks. *The table describes each of the microbenchmarks used to evaluate VMM-MemRx.*

techniques may be required to provide Geiger-like functionality on the latest VMM-aware architectures.

5.7 Summary

In order to handle modern operating systems containing unified system caches and journaling file systems, Geiger contains a number of sophisticated inferencing techniques. On microbenchmarks, Geiger is highly accurate as measured by eviction counts and one-to-one eviction accuracy; however, for write-intensive workloads, Geiger does experience a significant lag in when it detects that an eviction has occurred. On application workloads, Geiger also does well. For some swap-intensive workloads, additional techniques may be required to avoid detection of false evictions due to the difficulty of tracking liveness within swap. We have shown that Geiger's full range of techniques are needed under different circumstances: COW and Allocation techniques are needed to handle the Mogrify and TPC-W workloads given a unified buffer cache; live block detection improves the accuracy of delete-intensive workloads; finally, writes to the journal must be isolated to handle file system data journaling.

6. Case study: Working Set Size Estimation

The eviction detection techniques of Geiger are useful for implementing a number of pieces of functionality. In our first case study, we show how Geiger can be used to implement MemRx, a VMM that tracks the working sets of its guest VMs. We begin by describing the implementation of MemRx and then present performance results.

6.1 MemRx Implementation

Previous research by Waldspurger [38] for ESX Server has shown how a VMM can determine the system working set size of a VM whose memory footprint fits in physical memory. MemRx complements the ESX Server technique by enabling a VMM to determine the working set size for a thrashing virtual machine.

MemRx accomplishes this using Geiger to observe the evictions and subsequent reloads from the guest operating system buffer cache. MemRx then quantifies the number of memory accesses that would be transformed from misses into hits for several different memory sizes. MemRx simulates the page cache behavior of a virtual machine for each memory increment using a method similar to Patterson *et al.*'s ghost buffering [27].

Figure 9 shows a schematic of the page cache simulation implemented by MemRx. When a page is evicted, a reference to the page's location on disk is inserted at the head of a queue maintained in LRU order by MemRx. Subsequent evictions push previous references deeper in the queue. When a previously evicted page is read from disk, the reference to that page is removed from the queue and its distance (D) from the head of the queue is computed. The distance is approximately equal to the number of evictions that have taken place between that page's eviction and its subsequent reload. MemRx then uses (D) to compute the amount of memory that would have been required to prevent the original eviction from taking place as $size_{page} \times (D+1)$. This information is used to compute a miss-ratio curve. The working set size can be read from the miss-ratio curve by locating the curve's primary knee.

6.2 Evaluation

We first evaluate the accuracy of MemRx by using it to measure the working set size of microbenchmark workloads for which the working set size is approximately known. Table 10 lists each of the microbenchmarks and the actions they perform; the working set size for each is approximately 256 MB and the virtual machine is configured with 128 MB of memory. Second, we compare the working set size predicted by MemRx to the working set size determined by trial and error for more realistic application workloads, in particular, Mogrify and Dbench.

Figure 11 shows the predicted and actual miss ratio curves for the four microbenchmark workloads. The miss ratio curve shows the fraction of the capacity cache misses occurring in the smallest memory configuration (*i.e.*, 128 MB) that remain misses in larger memory configurations. The *predicted* curve is calculated by MemRx using measurements taken during a single run at the smallest memory configuration and then simulating the page cache behavior of the guest operating system on-line for several larger memory configurations in increments of 32 MB. The *actual* curve is calculated by running the workload at each of the noted memory sizes and counting actual capacity misses in the page cache.

These calibrated tests show that MemRx can locate the working set size of simple workloads very accurately. The prediction made by MemRx is identical to that found by direct measurement using trial and error. The result is not surprising, because under these simple workloads, Geiger incurs few eviction false positives.

Figure 12 shows the results for the two application workloads, Mogrify and Dbench. The leftmost two graphs show the predicted and actual miss ratio curves. In these cases, the inferred working set size predicted by MemRx is slightly larger than the actual working set size found using trial and error. To determine whether the discrepancy was due to Geiger (*e.g.*, false positive/negative evictions or lag) or to MemRx (*e.g.*, cache simulation error) we implemented MemRx within Linux and compared the predicted and actual miss ratio curves produced by that version. Within the operating system, MemRx has access to precise eviction and promotion information, which eliminates Geiger as a source of error. The rightmost two graphs in Figure 12 show the miss ratio curves obtained for the Mogrify and Dbench workloads using our operating system implementation of MemRx.

For the Dbench workload, the version of MemRx in the OS shows the same deviation as the one produced by MemRx in the VMM; this leads us to conclude that the cause of the deviation is MemRx simulation error. MemRx models the guest buffer cache using a strict LRU policy that does not exactly match the policy used by Linux, which is something more akin to 2Q [21]. The difference between the modeled policy and the true policy leads to simulation errors like the one shown. In the case of Mogrify, however, the OS-based miss ratio curve matches the actual curve closely, leading us to believe that the error observed in the VMM-predicted working set size is due to the small inference errors imposed by Geiger and the granularity of the experiment.

In summary, the information provided by Geiger is useful for enabling a VMM to estimate the working set sizes of thrashing VMs. The predictions made by MemRx are accurate enough to be highly useful when allocating memory between competing VMs on a single machine or when selecting an appropriate target host during virtual machine migration.

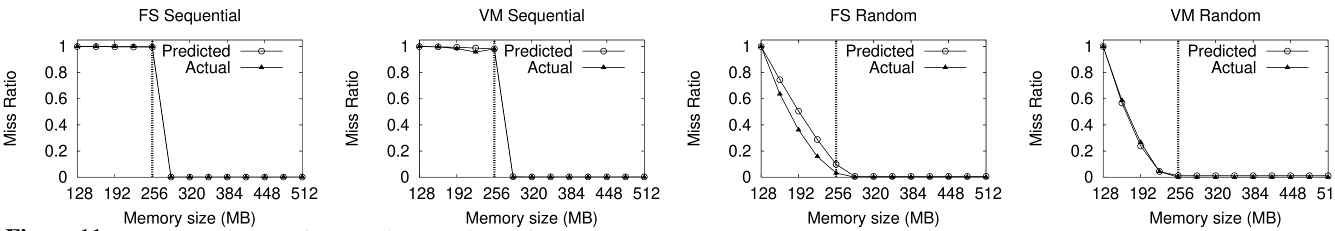

Figure 11. VMM-MemRx Predicted vs. Actual Miss Ratio. *The figure shows the miss ratio predicted by VMM-MemRx vs. the actual miss ratio measured for varying memory sizes. The known working set of 256 MB is marked by a vertical dashed line.*

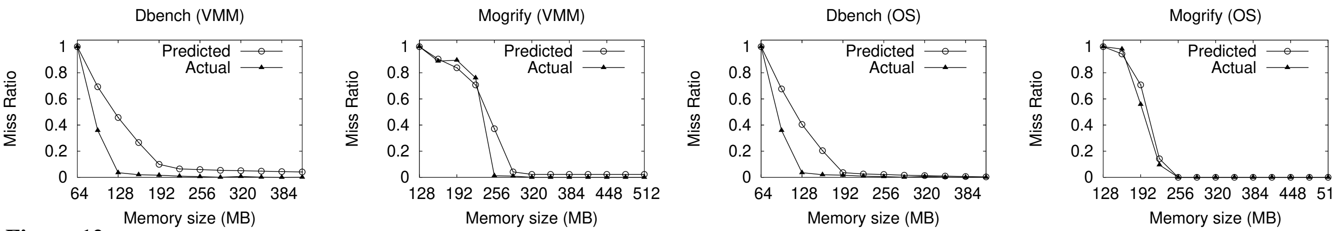

Figure 12. Application Predicted vs. Actual Miss Ratio. *The figure shows the miss ratio curve predicted by MemRx vs. the actual miss ratio measured for varying memory sizes for two application workloads. Results from MemRx implemented in the VMM (left) and MemRx implemented in the OS (right) are shown.*

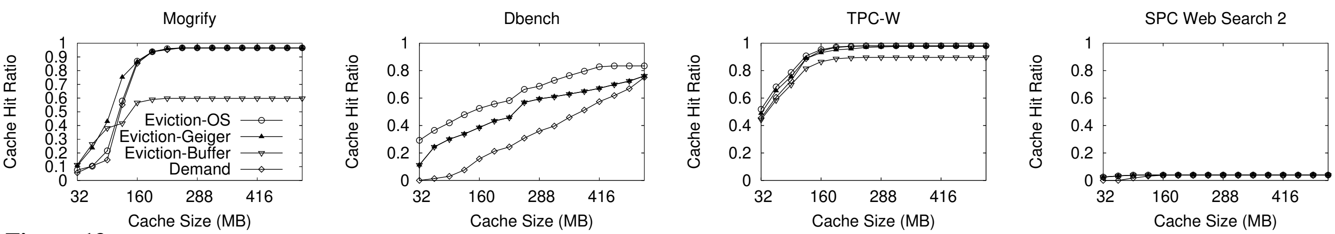

Figure 13. Secondary Cache Hit Ratio. *The figure compares the cache hit ratio in a secondary storage cache for various workloads when demand placement (Demand), eviction placement based on inferred evictions (Eviction-Buffer and Eviction-Geiger), and eviction placement based on actual evictions (Eviction-OS) is used. Experiments are performed using cache sizes from 32 MB to 512 MB.*

7. Case study: Eviction-Based Cache Placement

In our second case study, we show how Geiger can be used to convey eviction information to a secondary cache. The basic idea is that the VMM uses Geiger to infer which pages have been evicted from the OS buffer cache, then sends this information (*e.g.*, with a DEMOTE operation [40]) to the storage server, which is potentially remote. The storage server uses this explicit information to perform eviction-based cache placement.

7.1 Implementation

Our implementation of an eviction-based secondary cache has two components. First, the VMM interposes on the virtual block device interface provided by Xen to see the block request stream generated by the workload. Second, the VMM uses Geiger to infer which blocks have been evicted from the VM page cache; these events are then communicated to the remote storage server. We simulate the behavior of a storage server by using the actual trace gathered from running Geiger for a given workload as input. We refer to our approach as Eviction-Geiger.

To evaluate our implementation, we compare to three alternatives. In the first approach (Eviction-OS) the operating system is modified to report actual evictions; this represents the ideal case. In the second approach (Eviction-Buffer), the VMM performs only the eviction detections that are possible using client buffer addresses as used by Chen *et al.*(*i.e.*read and write evictions). Finally, we simulate a storage cache that uses no information about client evictions and performs traditional, demand-based placement. In all cases we use an LRU-based replacement policy.

7.2 Evaluation

We use the application workloads listed in Figure 2 to evaluate our VMM-implementation of eviction-based cache placement. For each workload, we consider remote caches from 32 MB to 512 MB. We evaluate the four placement policies: Eviction-OS, Eviction-Geiger, Eviction-Buffer, and Demand. Our metric is cache hit ratio.

Figure 13 shows graphs of the cache hit ratio vs. cache size for the four workloads and four cache policies. In all cases, OS and Geiger eviction-based placement outperform demand-based placement, sometimes significantly. The largest gains occur for moderate cache sizes where the working set of the application fits neither in the client cache nor in the storage cache individually, but does fit within the aggregate cache. OS and Geiger eviction-based placement are able to improve cache hit rate by as much as 28 percentage points for these workloads. For example, under the Mogrify workload using a secondary cache size of 96 MB, the cache hit ratio goes from 14.9% under demand placement to 42.9% when eviction-based placement is used. When the secondary cache size is large enough to contain the full system working set, OS and Geiger eviction-based placement perform similarly to demand-based placement. In the case of SPC web search, the traces exhibit almost no locality. The results are included for completeness only.

For one workload, Dbench, eviction-based placement with OS support outperforms that with inferred evictions, even with Geiger. For example, with a secondary cache size of 416 MB, we observe a difference in hit rates of about 15 percentage points. This performance difference is due to the significant time lag between the actual and inferred write-eviction events (approximately 2 seconds for most events in this experiment). Because some inferred evictions are delayed, the secondary cache loses the opportunity to place a block prior to the block being referenced by the client, and a cache miss occurs. However, the eviction-based approaches still perform significantly better than demand-based placement.

Eviction-Geiger always performs as good or better than Eviction-Buffer. In fact, Eviction-Buffer sometimes performs significantly *worse* than straight-forward demand-based placement. The problem occurs because Eviction-Buffer may detect fewer evictions than actually occur (*i.e.*, large false negatives). For workloads, such as Mogrify and TPC-W, where a significant number of non-I/O based evictions occur, missing evictions lead to poorer overall cache performance. Missing evictions are particularly a problem with large secondary caches, because few blocks are placed effectively, even though adequate cache space is available. In the case of TPC-W, missing eviction events change the cache hit rate by about 10 percentage points, while under Mogrify the difference is about 40 points.

In summary, Geiger can be used effectively to notify a secondary cache of the evictions that have been performed by clients. As confirmed in other studies [9, 8, 40], secondary caches using eviction-based placement can perform much better than those using demand-based placement. Our results show that the eviction information provided by Geiger is nearly as good as that which could be provided directly by the OS (if the OS were modified to do so). The one exception occurs when a significant lag occurs in the time between the actual eviction and the inference; however, even in this case, Geiger enables much better hit rates than those with simple demand-based placement. With eviction-based placement, it is essential to not miss evictions in the clients; eviction detection based only on I/O reads and writes can miss important evictions, leading to hit rates that are actually worse than simple demand-based placement. Therefore, the full set of techniques within Geiger should be used for buffer cache inferences.

8. Conclusion

On backend servers and desktop PCs, virtualization is becoming commonplace. As the virtual machine monitor becomes the sole resource manager of a system, many pieces of interesting functionality will likely migrate from the operating system into the VMM. The key to enabling new VMM-level functionality is *information*, *i.e.*, knowledge of OS-level constructs that are typically needed to implement various features.

In this paper, we have explored the techniques required to make inferences about when pages are added to or removed from the OS buffer cache. We have found that certain key features of modern operating systems cannot be ignored, including unified buffer caches and virtual memory systems, journaling file systems, and disk block liveness. Overall, our techniques are efficient, and allow us to implement two useful prototype case studies: a working set size estimator and eviction-based cache placement for second-level caches.

Inferring information across the boundary between a VMM and its guest operating systems is a powerful technique that enables new systems innovation to be implemented portably within a VMM. The more accurate, timely, and general these techniques can be made the more likely they can be successfully applied in the commercial domain.

Acknowledgments

We would like to thank the anonymous reviewers for their thoughtful comments and suggestions. This research is sponsored by the Sandia National Laboratories Doctoral Studies Program, by NSF CCR-0133456, ITR-0325267, CNS-0509474, and by generous donations from Network Appliance and EMC.

References

[1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[2] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

[3] S. Ballmer. Keynote address. Microsoft Management Summit, April 2005.

[4] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[5] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 1–11. ACM Press, 1995.

[6] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[7] P. M. Chen and B. D. Noble. When Virtual Is Better Than Real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133. IEEE Computer Society, 2001.

[8] Z. Chen, Y. Ahang, Y. Zhou, H. Scott, and B. Schiefer. Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems. In *Proceedings of the 2005 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, Banff, Canada, June 2005.

[9] Z. Chen, Y. Zhou, and K. Li. Eviction-based Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 269–282, San Antonio, Texas, June 2003.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[11] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[12] P. J. Denning. Working Sets: Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[13] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[14] R. Figueriredo, P. Dinda, and J. Fortes. A Case for Grid Computing on Virtual Machines. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[15] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *OASIS ASPLOS 2004 workshop*, 2004.

[16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

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

[18] P. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[19] ImageMagick Studio LLC. ImageMagick image processing software. http://www.imagemagick.org.

[20] Intel Corporation. Intel Virtualization Technology Specification. ftp://download.intel.com/ technology/computing/vptech/C97063.pdf, 2005.

[21] T. Johnson and D. Shasha. 2Q: A Low-Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 439–450, Santiago, Chile, September 1994.

[22] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, Boston, Massachusetts, June 2006.

[23] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, pages 364–373, Seattle, Washington, May 1992.

[24] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[25] D. Muntz and P. Honeyman. Multi-Level Caching in Distributed File Systems - or - Your Cache Ain't Nuthin' But Trash. *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.

[26] Open Source Development Labs. OSDL Database Test Suite. http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite.

[27] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, Colorado, December 1995.

[28] H. Reiser. ReiserFS. www.namesys.com, 2004.

[29] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 377–390, Boston, Massachusetts, December 2002.

[30] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[31] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the Third USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, CA, March 2004.

[32] Storage Performance Council. SPC web search engine storage traces. http://traces.cs.umass.edu/storage.

[33] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[34] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

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

[36] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[37] S. C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/ release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[38] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[39] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[40] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[41] M. Zhao, J. Zhang, and R. Figueriredo. Distributed File System Support for Virtual Machines in Grid Computing. In *Proceedings of High Performance Distributed Computing (HPDC)*, July 2004.

[42] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 91–104, Boston, Massachusetts, June 2001.