Design Choices for Utilizing Disk Idleness in a Virtual Machine Environment

Pradheep Elango, Saisuresh Krishnakumaran, Remzi H. Arpaci-Dusseau
{pradheep, ksai, remzi}@cs.wisc.edu
Computer Sciences Department
University of Wisconsin, Madison

May, 2006

Abstract

In virtualized environments, an operating system may not have complete knowledge about its resources, as it sees only virtualized forms of physical resources. The Virtual Machine Monitor which has access to the physical resources, however, is not aware of the abstractions of the operating system. In this paper, we discuss how this lack of information can hinder the implementation of certain mechanisms that require both kinds of information. Specifically, we address how information about disk idleness can be passed to virtual machines so that idle disk periods can be effectively utilized to maximize disk bandwidth. The main focus of this paper is the discussion of various mechanisms that could be applicable in a virtualized environment in order to effectively expose such information and exercise control. We discuss designs to infer the number of dirty pages in each domain from the VMM, and to coerce a domain to flush its dirty pages. Finally, we present an evaluation of our approaches.

1 Introduction

A Virtual Machine (VM) provides a software environment that can encapsulate one or more operating systems. Virtual machines (VMs)[3, 6, 7, 15, 16] bring in advantages that include server consolidation, support for legacy applications and multiple operating systems, sandboxing, fault tolerance and others. While virtualization naturally induces processing overhead, recent hardware advances such as the Intel Vanderpool technology and software advances such as paravirtualization have minimized the CPU performance overhead of running virtual machines.

Virtual machines add an extra layer of abstraction, providing a virtualized view of physical resources to the OS. Abstraction leads to hiding away some information. In a virtualized environment, an operating system does not have complete knowledge about its resources. The operating system is not the all powerful resource allocator as it is in the normal environment because it sees only virtualized resources. Therefore, it is typically unaware of the immediate availability of the resources. This lack of awareness can prevent an operating system from taking informed decisions about resource allocation. The more privileged virtual machine monitor (VMM) can directly observe and control the physical resources. However, application-specific information that are generally available to an OS are not available to a VMM, that prevents it from executing informed resource scheduling.

In this paper, we investigate the lack of awareness of operating systems running on virtual machines, with respect to disk availability. Generally, in a disk system, write requests to the disk are batched by the disk driver, whereas reads happen synchronously. While this could amortize the overall write latency, it will also affect the read requests that arise during flushing. Thus, the latency of reads could be affected depending on their timing; and the frequency and size of the write requests. One simple approach is to schedule delayed writes whenever the disk becomes idle [12], rather unconditionally. We refer to this strategy as *opportunistic flush*. To perform an opportunistic flush, the operating system requires knowledge of the disk idleness. Virtualized environments provide a challenge to an operating system to obtain this information, as the OS can observe only the virtualized disk. In this paper, we discuss different challenges involved in communicating disk idleness to an OS running on a VM. For the sake of simplicity, we do not consider other factors such as predicting the length of idleness in order to take a decision.

Opportunistic flush gives rise to another challenge in a virtual machine environment. Multiple domains (that is, operating systems on VMs) could be containing different amounts of dirty data. So, in order to make the best use of the idle disk, the virtual machine monitor (VMM) should schedule the domain with the maximum amount of dirty data. A straightforward way would be for the OS to explicitly transfer such information to the VMM; however, this is not always possible as the OS could be proprietary. Further, if a VMM can observe and infer specific information, the code needs to be implemented only once which will benefit operating systems on other virtual machines above. In this paper, we present a method that exploits the virtual machine architecture in order to observe the *dirtiness* of an OS. Moreover, we present

a method for the VMM to schedule a chosen domain, and discuss loose control mechanisms (that do not require any code changes), and stronger mechanisms (which need code changes) in order to implement these policies.

In Section 2 we describe related work. We discuss our design in Section 3, and provide implementation details and issues in Section 4. We finally present an evaluation of our methods in Section 5 and conclude in Section 6.

2 Related Work

The main advantages of using a delayed write mechanism are the effects of write cancellation, burstiness and efficient utilization of disk bandwidth [5].

The idea of using delayed writes has been studied in the past [4, 12] and has been employed by many popular operating systems such as Unix and its various flavors. In [12], Mogul observes that technology trends in memory and disk vary a big deal. While the increasing trends of DRAM technology has led to an effective increase in main memory size, effectively increasing the size of the "buffer cache" for a disk, disk access times have not kept up by the same standards. This disparity results in an increase in the time required to flush all buffers in buffer cache to the disk. Mogul proposes a new technique called Interval Periodic Update policy, which is similar to the normal periodic update policy except for the fact that during every periodic flush, only buffers that are aged above a threshold age are flushed. He also suggests that this may still not solve the problem of long queues. In order to avoid implementation of complex timers, he proposes an Adaptive Interval Periodic Update (AIPU) policy. In fact, flavors of the AIPU policy finds its place in many modern operating systems such as Linux.

More recently, in [13] and [14] the importance of using idle time that is widely prevalent in I/O workloads today, is discussed. In [14], the authors infer that bursty patterns can be created to conserve power, thereby maximizing disk utilization. Arguing that the normal AIPU policy will lead to frequent idle time intervals of 5 seconds or less, they simply change the frequency of the update daemon to flush all dirty buffers once every minute. However, their approach involves many changes to the OS such as certain parts of the file system interface itself, in order to provide hints to the OS. Thus, in a way, they do the opposite of what we are trying to do, since their approach increases burstiness (instead of spreading it out) so that mean length of idle intervals can be increased in order to provide significant energy savings. Our work is different in that our main goal is to increase disk bandwidth utilization (rather than power conservation). However, our framework will still be applicable if the same problem needs to be addressed in a virtualized environment as our mechanisms help expose information that can be used to exercise control for different purposes.

In [9], Hsu and Smith study the behavior of common workloads on personal computers and server systems. The authors study I/O behavior at a physical level rather than at the logical level and observe that there is much interaction between reads and writes, and they seldom tend to happen in isolation. Therefore, they strongly argue that idle periods in disks can be utilized for background operations, such as block reorganization. In this paper, we discuss how we can use it for flushing delayed writes.

Golding et al. give a primer on the usefulness of idleness in [8]. They present an extensive study into how idleness can be useful in computer systems, and how one can effectively use idle processor cycles and idle time intervals available in disks. Further, it discusses typical designs of systems that use idle time, and stresses on the importance of predicting idle time intervals, and the value of predictions and feedback in the system.

In this paper, we take the standpoint that knowing about the idleness of the disk can be very useful, and we discuss how we can build mechanisms to control the behavior of an operating system in a virtualized environment.

A virtualized environment hides information about physical devices and therefore is philosophically opposite to Exokernel [11] systems which strongly argue that all hardware resources must be directly exposed to operating systems doing away with all abstractions. More practical research on extensible operating systems such as gray-box systems [1] and infokernel [2] systems emphasize the importance of minimizing changes to the OS code base. While graybox systems strictly view the OS code to be unchangeable, infokernel systems allow OS code to be modified, but just to expose information. In a virtualized environment, therefore, extending the OS with additional services to the VMM to expose information will constitute an infokernel approach, whereas inferring the OS state from the VMM without modifying the OS will be a graybox approach. Since the more privileged monitor runs beneath the OS, there could be more opportunities to infer about the OS state because of possible traps to do "privileged" operations. Though, in a sense the monitor is actually the "operating system", modifying the monitor code is easier and could be more advantageous than modifying the code of operating systems on top of it. Techniques to gather information about OS abstractions and to exercise control on them from beneath the OS is the topic of current research [10].

3 Design

Figure 1 shows the overall architecture of our framework. The host operating system detects disk idleness by tracking the active disk requests. The host OS then calls the virtual machine monitor (VMM) and communicates the disk idleness information. The VMM then either calculates the number of dirty buffers in each of the guest operating systems or uses precalculated values and schedules the guest with the maximum dirty buffers. The VMM then coerces the guest operating system to flush its dirty buffers. This effectively reduces the effect of burstiness during periodic updates. In the following

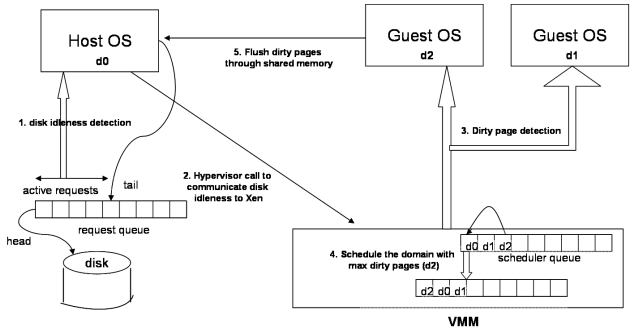

Figure 1: **Overall design:** *This figure illustrates the various events that trigger information exposure, and the general flow of information and control in our framework*

subsections, we discuss the design of each of these issues in greater detail.

3.1 Detecting Disk Idleness

I/O virtualization makes detection of disk idleness from the guest operating systems difficult. In a virtualized environment, the host OS is the single location where all the disk requests arrive and the guest operating systems typically sees virtualized resources. Hence information about disk idleness has to be passed from the host OS. The host OS can export information about the number of requests currently in the queue of any particular disk similar to the disk idleness detector, InfoIdleSched [2]. The main difference here is that the OS passes information to the VMM which is a more privileged layer.

3.2 Detecting Dirty Buffers

One of the key features in this system is the ability to detect the number of dirty buffers without actually changing the guest OS code. This information is necessary for the VMM to schedule the guest OS that can utilize the idle disk. We outline a few possible approaches below.

3.2.1 Installing a driver

A driver installed in the guest operating system can serve to pass the information about the number of dirty pages. Alternatively, a list of dirty pages ordered by their relative likeliness to go to disk, can be exported by the operating system to the VMM similar to the abstractions developed in Infokernel [2]. The principal limitation of this approach is that each operating system must now be supplied with this driver.

3.2.2 Page Table Walking

Dirty pages can also be detected by traversing the page tables of the virtual machines. We assume that we have "gray-box" knowledge of the structure of the page table used by a virtual machines. A VMM typically has to ensure that any particular guest OS can modify only the page table entries that the guest owns. Thus, it is necessary to trap all page table entry updates in order to keep track of the pages owned by each virtual machine. This technique can be fine tuned by keeping track of all page table creates and deletes instead of page table entry updates, and then traversing only through this list to find all dirty pages. This mechanism not only helps find the number of dirty pages in a virtual machine, but also the actual machine addresses of pages that are dirty.

3.2.3 Page Sampling

The number of dirty pages can also be estimated using sampling techniques. The choice of sampling method plays a significant role in the accuracy of sampling and sampling methods can be as simple as selecting a small random number of pages. A more relevant set of sample pages can be maintained by keeping track of all the page table creates and updates which gives a better estimate of the dirty pages. An obvious limitation of sampling, however, is that one can only expect an estimate of the total number of pages to be flushed.

3.2.4 Predicting from host OS

Another approach to detect the number of dirty pages is to observe the I/O requests passing through the communication channel between the host OS and a guest OS. This information can be used to predict the number of dirty pages a guest OS will have on its subsequent periodic updates. This assumes that the main write traffic generators are the guest operating systems. While this technique is less heavy-weight, it could suffer a drop in accuracy and currency of the information.

3.2.5 Checksumming to track dirty pages

Checksumming can also be employed to estimate the number of dirty pages in a virtual machine. The checksum of every page in memory is maintained and a page is predicted as dirty if its checksum changes. The advantage of this approach is that it does not rely on the dirty bits present in the page table entries. However, the entire page has to be read and processed in order to calculate the checksum. Thus, checksumming has a higher overhead compared to reading the page table entries directly. With checksumming it is also not possible to differentiate whether a page has been dirtied or newly read. Since newly read pages could possibly replace a page from a page frame, the checksum for a particular address will change.

3.3 Coercing domains to flush the dirty pages

Once the VMM knows that the disk is idle and the pending write load of the different domains, it needs to select a domain to schedule, and effectively control it to flush its requests (to its virtualized disk), and then schedule the host OS so that these requests are actually transferred to the real disk, thereby

utilizing the idleness. We outline possible mechanisms to coerce a selected domain to flush its dirty pages.

3.3.1 Pull-based Mechanism

In order to flush the dirty pages of a user domain transparent to the OS running in that domain, we could use the information available in the VMM about the dirty pages present in a particular domain and write them to disk when the disk is idle. In order to avoid flushing the same pages, we need to track information about the pages that are written to the disk. Pages that are written this way can be dirtied once again by the guest OS, and in such cases, they need to be flushed again. A simple way to do this would be to compute the checksum of the pages we write ahead during idle periods. When the OS actually flushes the pages, we compare the checksums and decide whether to write the page back to disk. We write back those pages for which the checksums do not match. However, when the checksums do match, one could choose a cautious approach, and store page state also along with the checksums, in which case, the entire page contents have to be compared whenever the checksums match. As an alternate optimistic approach, one could avoid storing the contents totally, and not flush when the checksums match. For this, we need strong properties in the function that generates the checksum.

3.3.2 Forcing the early scheduling of the update daemon from the VMM

Exercising control over an operating system scheduler policies, without actually changing the OS is a challenging problem. If the OS exports an abstraction, which indicates the time remaining for its update daemon to be scheduled next, then the VMM could use this information in order to make its decision, and select a domain whose update daemon is more likely to run. If the OS can export an abstraction that even lets the time interval for an update daemon to be modified, then the VMM can reduce this interval to its lowest limit, and then schedule the domain.

3.3.3 Using a driver that calls the update daemon

Alternately, a driver could be inserted into the user domain which communicates with the VMM and just calls the update daemon when instructed by the VMM. Again, the issues here are portability of the driver to different operating systems. Modifying the driver essentially is the same as modifying the operating system, similar to the OS abstraction discussed above.

3.3.4 Using a balloon driver

Sometimes, virtual machine environments provide balloon driver interfaces, that let a domain to dynamically control the size of memory allotted to a virtual machine. Generally, a balloon driver is used to reclaim memory pages of an OS so that the policies of the OS are used when choosing pages to be swapped out [16]. In this situation, the VMM will instruct the balloon driver to inflate. However, a balloon driver would pin most of the clean pages when it exerts memory pressure on the OS. This may not be the best way to flush a domain's dirty pages because directly using it does not ensure that all dirty pages will be flushed. For this to happen, the VMM could increase the balloon pressure by demanding more and more memory, and keep track of all the pages eventually restoring back all pages that were claimed. However, the overheads of this mechanism need to be investigated.. Further, memory limitations on drivers/modules could make this technique infeasible.

4 Implementation

We use Xen 2.0.5 as our base platform for implementing our design. Xen is a virtual machine monitor (VMM) that supports execution of multiple guest operating systems (user domains). All I/O is managed by a privileged operating system called the host OS (Domain 0) which has complete access to the physical resources. Figure 2 shows a typical I/O path in the Xen environment. A guest OS sees only virtualized resources, and requests are communicated through a shared memory channel to the host OS.

The following sections provide details of certain mechanisms that have been implemented in this framework.

4.1 Idleness detector

This section addresses the issues of detection of disk idleness at the host OS and communication of this information to the host OS.

User domains communicate to Domain 0 via a shared memory channel. This privileged domain is the only place where the actual disk idleness can be determined accurately. The disk idleness information can either be communicated to the Xen hypervisor (VMM) or to the user domains. Communicating the idleness information to the user domains will require changes to the user domain OS code (or the blockdriver portion of it). This also makes the control over scheduling of a particular domain difficult. Hence communicating the disk idleness information to Xen would be the best possible choice because the user domain OS code need not be modified and VMM will also have a global knowledge of the state of dirty buffers in all the user domains. The following sections describe finer details of communicating the idleness to the VMM.

Detecting disk idleness in Domain 0: Domain 0 is the single point through which the entire disk traffic flows. We could detect disk idleness using one of two ways. One way is to read the disk queue directly and determine the number of active requests. The other way is to track the number of requests that pass across the file system / block driver boundary.

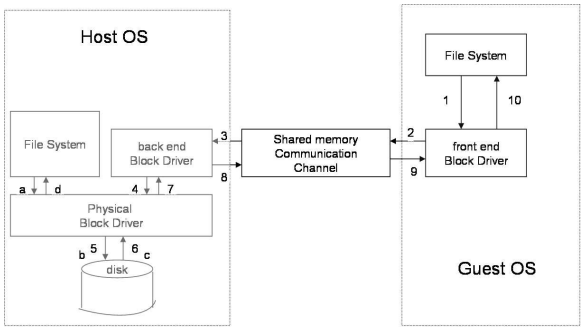

Figure 2: **I/O path in Domain 0 and Domain 1.** *a,b,c,d shows the order of events that occur during I/O operation in the host operating system. 1 - 10 shows the order of events that occur during I/O operation in the guest operating system.*

Figure 2 shows the path followed by disk requests in Domain 0 and Domain 1.

We maintain a counter, which is incremented every time `submit_bio()` is called and decremented every time `end_bio()` is called. `submit_bio()` is the interface through which block I/O requests are submitted to the driver, and `end_bio()` is the interface through which driver notifies the OS of the completion of the I/O request. The counter at any time reflects the number of active disk requests. We declare the disk is idle if the number of active requests falls below a threshold. Disk idleness is communicated to Xen by a hypercall.

4.2 Detecting Dirty Pages

While kernel page tables are stored and pinned in fixed memory locations, processes create page table dynamically, in the Linux operating system. So, in order to keep track of all dirty pages, one heavy-weight way will be to go through all the machine pages that are allocated to a domain. The Xen data structure *pfn_info* includes all meta-information about every machine page that is allotted to a domain. We use this array to locate dirty machine pages.

4.2.1 Page Table Walking

One way to count all the dirty pages in a system would be by calculating the number of dirty page table entries in the entire OS. For a given domain, we count the number of dirty page table entries in each Level-1 page table page.

A limitation with this approach is that the time taken to traverse the entire range of pages could turn out to be prohibitive, and it could trigger any watchdog timer to reboot the system.

4.2.2 Page Table Sampling

Instead of traversing through all pages, we can traverse through a sampling of pages. We implemented a stride sampling technique; instead of choosing the immediately next page for traversing, we choose the page that is "stride" units away. This effectively divides the total number of pages that we traverse by "stride".

By randomly changing the starting point of traversal and the stride length, we can effectively traverse the entire list of pages. This provides a way to quickly carry out the counting work.

4.2.3 Detecting number of dirty pages with a driver

In Linux, much of the OS state can be tracked using the `/proc` filesystem. For example, `/proc/meminfo` can provide information about the number of dirty file pages. Therefore, one way for the hypervisor to get the number of dirty pages will be by installing a driver in the guest OS, that will directly export this information to the Xen hypervisor. As an optimization to reduce the communication overheads between the driver in the guest OS and the hypervisor, the driver can directly pass the machine address of the location in (kernel) memory that keeps track of the number of dirty pages. The hypervisor then directly maps this address to its address space, and can read whenever required, thus eliminating many context switches.

One important limitation with this approach however is that these counters do not include all pages that have to be written back. For example, memory-mapped pages that are dirtied will not be kept track of.

4.3 Scheduling the domain with maximum dirty pages

In our implementation, we modified the round robin scheduler to schedule the domain with maximum dirty pages. The round robin scheduler chooses the domain at the head of the ready list to schedule next. Once we receive a hypercall from Domain 0, we traverse the ready list looking for the domain with maximum number of dirty pages. We then move the domain to the head of the list and call the scheduler routine that starts executing this domain. This however will not be enough if we want the writes to be flushed as soon as possible. We further place Domain 0 just behind the selected domain, so that the requests can be written to disk as soon as possible. Alternatively, each domain structure has a time slice field which is set to the time quanta of round robin scheduler each time the domain is scheduled. Thus, we could have simply changed the time quanta of all the domains that are ahead of this domain in the ready list to zero for that iteration. This change would have made that particular domain to be scheduled next. However, there is significant overhead associated with this approach, where the scheduler has to change the context for each of these domains only to realize that their time quanta has expired. Further, these domains will lose a scheduling opportunity; so, this method is a bit unfair.

4.4 Tracking file-backed pages

Counting dirty bits in all page table entries does not give all the dirty pages in an OS. Specifically, file-backed pages will not be reflected accurately by just looking at the page table entries. File-backed pages are those pages that are mapped from a file to the memory. Changes to the memory page will be reflected in the file, when the page is flushed out to the disk.

In Linux, file-backed pages are flushed by a daemon process. File-backed pages are all stored in the page cache in the kernel address space; these pages actually do not "belong" to any process, and therefore are not mapped in the user address space.

Further, it is difficult to observe any file-backed pages being written to the disk, from below the OS. This is because, the periodic flush daemon makes use of the address space structure inside every process which has a list of dirty pages. The hardware is not at all used to detect dirty file pages. Thus, we cannot detect dirty file pages by counting all dirty page table entries in the page cache. Further, the dirty bits in these entries are *sticky*; so the OS does not reset the bits once they are set.

Our Approach: We reset all the dirty bits in all page table entries of the page cache from the hypervisor. In Linux, the kernel page directory is loaded at a fixed location. From this location, we traverse to access the page tables that belong to the page cache.

In order to estimate the number of dirty file-backed pages at any point of time, we reset the dirty bits in all these page table entries. After a short interval, we count the number of dirty bits in these page table entries. This effectively provides us a good estimate of the number of dirty file-backed pages. Since the kernel does not use these dirty bits for any accounting or operational purposes, we can safely reset these dirty bits.

To evaluate this approach, we ran a program that issues sequential writes totaling a certain amount in an infinite loop. Using the above approach, we were able to determine the exact number of dirty file-backed pages that a domain has at a particular point.

4.5 Exercising control over the update daemon from the VMM

The `/proc` file system interface in Linux exports information such as time remaining for "dirty writeback" and the threshold for dirty-to-clean pages ratio to trigger writeback. Since it provides a read-write interface, the Xen VMM can directly overwrite the parameter which determines when `pdflush` is called in the user domain. Thus, once the domain is scheduled, `pdflush` would be transparently scheduled by the OS, which would in turn flush the dirty pages. Therefore, in order to let the VMM exercise control over the `pdflush` daemon OSes at least need to export the memory location of these entries which will be then mapped by the hypervisor into its address space.

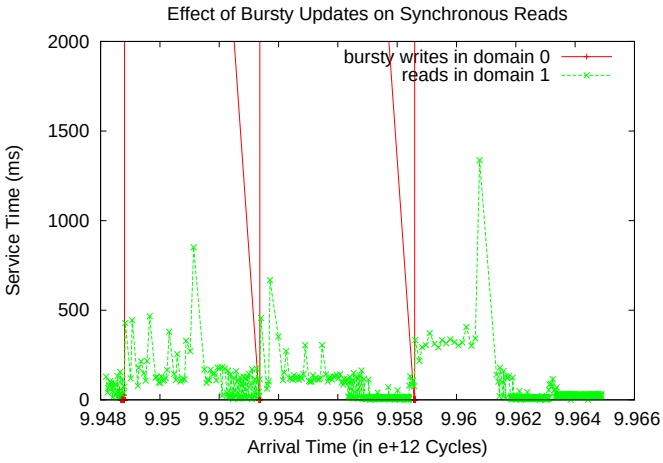

Figure 3: **Effect of bursty updates on read requests.** *Periodic writes issued from the host OS can affect the performance of reads in a user domain.*

During the course of our implementation, we observed that `wb_kupdate()` (the function triggered by `pdflush`) is not guaranteed to flush immediately because it might be sleeping waiting on some locks. We thus invoke `emergency_sync()`, which is more responsive, inside the domain. The unprivileged domains receive software interrupts from Xen, whenever Xen knows that the disk is idle. The obvious downside of this approach is that it requires adding a driver (to handle the software interrupt) to each of the unprivileged domains.

5 Evaluation

In this section, we describe a few experiments that we used to test the efficacy of our system. We first present an experiment that illustrates the problem, then an experiment to show how our methods can improve the situation. We then describe experiments that illustrate the effectiveness of different methods to find the number of dirty pages.

Our system ran on a 425 MHz, Pentium-III processor. Our experiments were run with one privileged and one user domain. Domain 0 was configured with 50 MB, and the user domain with 52MB. The system ran on a Seagate hard disk of capacity 8.62 GB, with a speed of 5400 rpm and an average read time of 10.5 ms.

5.1 Effect of bursty updates on read requests

Figure 3 shows the effect of bursty updates on read requests. The request arrival time is shown along x-axis and the service time is plotted along y-axis. We designed the experiment to make the reads contend with writes during bursty updates. Our program in Domain 0 kept generating many asynchronous writes that are buffered and to simulate the effect of periodic flush, we perform `sync` once in every 1000

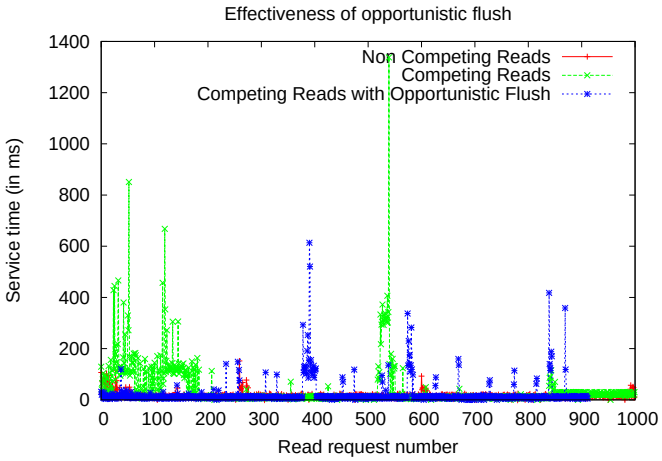

Figure 4: **Effectiveness of opportunistic flush.** *With opportunistic flush, the size of the peaks in the read times have vastly reduced.*

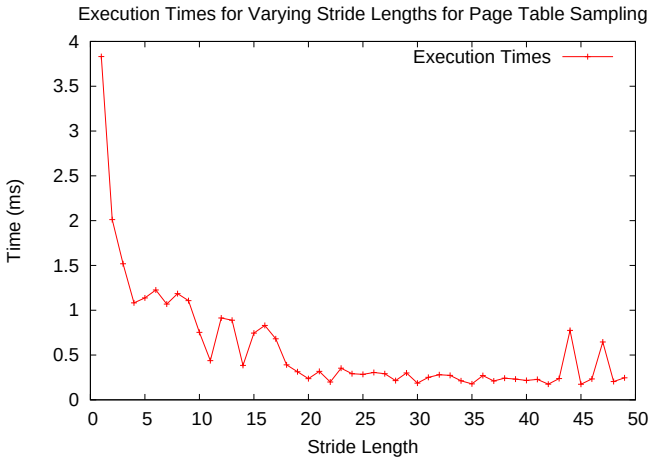

Figure 5: **Effect of stride length on total time for sampling.** *Time savings are not significant beyond a stride length of 20.*

writes. Another program generated continuous sequence of synchronous reads in the guest OS. The peaks indicate the service time for reads and they are greater because of the contention for the disk with the writes from Domain 0. If these writes are written to disk during disk idle times, it would have resulted in lesser contention during periodic flush and hence lesser service time for reads.

5.2 Effectiveness of opportunistic flush

Figure 4 shows the plot of service time versus the request number for three different situations. For non-competing reads, a program generated continuous read requests from the guest OS. For competing reads, we use the same program as described in 5.1. The peaks in read response time of competing reads are due to the contention with writes during the flush operations. Competing reads also suffer greater variance in service times during bursty write operations. The non-competing reads, on the other hand have a uniform service time. However, if we flush whenever the disk is idle (i.e. if we "opportunistically" flush), then we see that the peaks are much smaller. There are more number of peaks albeit smaller ones, because of the frequent disk flush operations. Opportunistic flush increases the average read bandwidth in these experiments from 42.33% to 80.87% of the read bandwidth in the non-competing case. Table 1 shows the overall benefit of this mechanism. Opportunistic flush improves read latency by nearly 50%, and throughput by nearly 55%.

5.3 Detecting dirty pages using sampling

In order to test the effectiveness and accuracy of our approach described in section 4.2.2, we conducted a few experiments.

5.3.1 Effectiveness of sampling

By increasing the stride length, we are reducing the effective number of pages that are processed. In our experiments, we varied stride lengths for sampling and measured relative error compared to complete traversal of all the page tables. In order to verify that stride sampling can really estimate the correct number of dirty pages, we increased the number of dirty pages by executing a program in the background which constantly kept dirtying data structures that were allocated on its stack, with fixed sizes. We observed that the accuracy of sampling can be unpredictable, with the error rising beyond 50% for stride lengths of over 20.

5.3.2 Effect of stride length on total time for sampling

Increasing the stride length means going through a smaller number of pages. This should reduce the total time for sampling. From Fig 5, we can observe that bigger stride lengths reduce the time taken, but after a certain threshold point, the savings in time is not significant enough.

We could strike a good trade-off between accuracy and time taken. In this case, it so happens that the point where we can get a good accuracy coincides with a point where we could get the best time savings.

5.3.3 Detecting dirty file-backed pages

In order to evaluate the approach described in section 4.4, we ran a program in the background that infinitely generated a fixed number of sequential writes. Since we did not want to include other dirty pages in the system, we first measured the number of dirty pages detected by running the program with no data being written. As we increased the number of pages that were written, we observed that we could get a very good estimate of the dirty file-backed pages. Table 2 shows the results of an experiment in which each trial was run 10 times.

	Read Response Time (ms)	Throughput (reads per sec)
Non Competing Reads	16.64	141
Competing Reads	39.30	60
Competing Reads with Opportunistic Flush	20.58	92

Table 1: **Analysis of Opportunistic Flush.** *The opportunistic flush strategy improves the response time and throughput of competing reads, and brings them closer to the response time and throughput of the non-competing reads.*

# Pages Written	# Dirty Pages Detected	Accuracy
0	115 (105-119)	-
25	141 (134-147)	96.0%
50	167 (154-169)	96.0%
75	191 (185-197)	98.7%
100	214 (202-220)	99.0%
125	241 (236-244)	99.9%
150	263 (257-266)	98.7%

Table 2: **Detecting dirty file-backed pages.** *This table shows the accuracy of our estimate for measuring the number of dirty file-backed pages. The second column indicates the total number of dirty pages in the system (with minimum and maximum number detected in our experiments inside parantheses), which includes those not written by our process.*

6 Conclusions

In this paper, we have discussed how the lack of information about physical resources in the guest OS and the lack of information about the operating system abstractions at the VMM can hinder the implementation of certain mechanisms. We have presented a basic framework in a virtualized environment, where knowledge about disk idleness can be effectively exported to other guest operating systems through the virtual machine monitor. We have discussed techniques to infer the number of dirty pages in a guest OS from the VMM. Our results show that opportunistic flushes can improve the read performance in presence of competing writes by about 50%.

Working below the OS on a more privileged platform, we can directly manipulate the hardware that the OS is running on. We have seen how we could make use of the hardware to convert a "silent" action into an observable event. Controlling scheduling from below the OS is not straight-forward, especially without communicating anything to the OS. In our implementation, a driver communicates to a kernel to achieve the task. The idea of controlling scheduling without changing the OS is challenging and needs to be explored more.

Acknowledgement

We thank Stephen Todd Jones and Vijayan Prabhakaran for the numerous discussions and unhesitated help they provided during our project. We also thank our anonymous reviewers for their comments on the initial draft of the paper.

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] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[3] E. Bugion, S. Devine, and M. Rosenblum. Disco:Running Commodity Systems on Scalable Multiprocessors. In *Proceedings of 16th Symposium on Operating Systems Principles*, Oct. 1997.

[4] S. C. Carson and S. Setia. Analysis of the Periodic Update Write Policy for Disk Cache. *IEEE Transactions on Software Engineering*, 18(1):44–54, Jan. 1992.

[5] P. M. Chen. Optimizing delay in delayed-write file systems. Technical Report CSE-TR-293-96, University of Michigan, May 1996.

[6] 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 ACM Symposium on Operating Systems Principles*, October 2003.

[7] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of the 2002 Symposium on Operating Systems Design and Implementation*, Dec 2002.

[8] R. A. Golding, P. B. II, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *USENIX Winter*, pages 201–212, 1995.

[9] W. Hsu and A. J. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2):347–372, 2003.

[10] 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 2006 Annual Technical Conference (USENIX '06)*, Boston, MA, June 2006.

[11] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *16th Symp. on Operating Systems Principles*, Saint Malo, France, 1997. ACM.

[12] J. C. Mogul. A better update policy. In *Proceedings of the 1994 Summer USENIX Technical Conference*, pages 99–112, Boston, MA, June 1994.

[13] A. E. Papathanasiou and M. L. Scott. Energy efficiency through burstiness. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'03)*, October 2003.

[14] A. E. Papathanasiou and M. L. Scott. Energy efficient prefetching and caching. In *In Proceedings of the 2004 USENIX Annual Technical Conference pages 255-268 Boston MA*, July 2004.

[15] S. Santhanam, P. Elango, A. Arpaci-Dusseau, and M. Livny. Deploying virtual machines as sandboxes for the grid. In *Second Workshop on Real, Large Distributed Systems (WORLDS 2005)*, San Francisco, CA, December 2005.

[16] C. Waldspurger. Memory resource management in VMware ESX server. In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, December 2002.