# Satori: Enlightened page sharing

Grzegorz Miłoś, Derek G. Murray, Steven Hand
*University of Cambridge Computer Laboratory*
*Cambridge, United Kingdom*
First.Last@cl.cam.ac.uk

Michael A. Fetterman
*NVIDIA Corporation*
*Bedford, Massachusetts, USA*
mafetter@nvidia.com

## Abstract

We introduce *Satori*, an efficient and effective system for sharing memory in virtualised systems. Satori uses *enlightenments* in guest operating systems to detect sharing opportunities and manage the surplus memory that results from sharing. Our approach has three key benefits over existing systems: it is better able to detect short-lived sharing opportunities, it is efficient and incurs negligible overhead, and it maintains performance isolation between virtual machines.

We present Satori in terms of hypervisor-agnostic design decisions, and also discuss our implementation for the Xen virtual machine monitor. In our evaluation, we show that Satori quickly exploits up to 94% of the maximum possible sharing with insignificant performance overhead. Furthermore, we demonstrate workloads where the additional memory improves macrobenchmark performance by a factor of two.

## 1 Introduction

An operating system can almost always put more memory to good use. By adding more memory, an OS can accommodate the working set of more processes in physical memory, and can also cache the contents of recently-loaded files. In both cases, cutting down on physical I/O improves overall performance. We have implemented *Satori*, a novel system that exploits opportunities for saving memory when running on a virtual machine monitor (VMM). In this paper, we explain the policy and architectural decisions that make Satori efficient and effective, and evaluate its performance.

Previous work has shown that it is possible to save memory in virtualised systems by sharing pages that have identical [23] and/or similar [4] contents. These systems were designed for unmodified operating systems, which impose restrictions on the sharing that can be achieved. First, they detect sharing opportunities by periodically scanning the memory of all guest VMs. The scanning rate is a trade-off: scanning at a higher rate detects more sharing opportunities, but uses more of the CPU. Secondly, since it overcommits the physical memory available to guests, the VMM must be able to page guest memory to and from disk, which can lead to poor performance.

We introduce *enlightened page sharing* as a collection of techniques for making informed decisions when sharing memory and distributing the benefits. Several projects have shown that the performance of a guest OS running on a VMM improves when the guest is modified to exploit the virtualised environment [1, 25]. In Satori, we add two main *enlightenments* to guests. We modify the virtual disk subsystem, to implement *sharing-aware block devices*: these detect sharing opportunities in the page cache immediately as data is read into memory. We also add a *repayment FIFO*, through which the guest provides pages that the VMM can use when sharing is broken. Through our modifications, we detect the majority of sharing opportunities much sooner than a memory scanner would, we obviate the run-time overhead of scanning, and we avoid paging in the VMM.

We also introduce a novel approach for distributing the benefits of page sharing. Each guest VM receives a *sharing entitlement* that is proportional to the amount of memory that it shares with other VMs. Therefore, the guests which share most memory receive the greatest benefit, and so guests have an incentive to share. Moreover, this maintains strong isolation between VMs: when a page is unshared, only the VMs originally involved in sharing the page are affected.

When we developed Satori, we had two main goals:

**Detect short-lived sharing:** We show in the evaluation that the majority of sharing opportunities are short-lived and do not persist long enough for a memory scanner to detect them. Satori detects sharing opportunities immediately when pages are loaded, and quickly passes on the benefits to the guest VMs.

**Detect sharing cheaply:** We also show that Satori's impact on the performance of a macrobenchmark—even without the benefits of sharing—is insignificant. Furthermore, when sharing is exploited, we achieve improved performance for some macrobenchmarks, because the guests can use the additional memory to cache more data.

The rest of this paper is organised as follows. We begin by discussing the issues of memory management in both operating systems and virtualised platforms (Section 2). We then survey related systems (Section 3).

We present Satori in two parts: first, we justify the major design decisions that differentiate Satori from other systems (Section 4), then we describe how we implemented a prototype of Satori for the Xen VMM (Section 5). Finally, we perform a thorough evaluation of Satori's performance, including its effectiveness at finding sharing opportunities and its impact on overall performance (Section 6).

# 2 Background

The problem of memory management has a long history in operating systems and virtual machine monitors. In this section, we review common techniques for managing memory as a shared resource (§ 2.1). We then describe the relevant issues for page sharing in virtual machine monitors (§ 2.2). Finally, we describe how paravirtualisation is used to improve performance in virtualised systems (§ 2.3).

## 2.1 Virtual memory management

Physical memory is a scarce resource in an operating system. If more memory is available, it can be put to good use, for example by obviating the need to swap pages to disk, or by caching recently-accessed data from secondary storage. Since memory access is several orders of magnitude faster than disk access, storing as much data as possible in memory has a dramatic effect on system performance.

Memory resource management was first formalised for operating systems by Denning in 1968, with the introduction of the *working set model* [3]. The working set of a process at time $t$ is the set of pages that it has referenced in the interval $(t - \tau, t)$. This is a good predictor of what pages should be maintained in memory. Pages can then be allocated to each process so that its working set can fit in memory.

Since it is challenging to calculate the working set and $\tau$ parameter exactly, an alternative approach is to monitor the *page fault frequency* for each process [16]. If a process causes too many page faults, its allocation of pages is increased; and vice versa. This ensures acceptable progress for all processes.

OS-level approaches are inappropriate for a virtualised system. One of the key benefits of virtualisation is that it provides resource isolation between VMs. If the size of a VM's working set or its page fault rate is allowed to determine its memory allocation, a malicious VM can receive more than its fair share by artificially inflating either measure. Instead, in our approach, we give a static allocation of physical memory to each VM, which provides strong performance isolation [1]. As we describe in § 4.2, our system provides surplus memory to VMs that participate in sharing. Our approach follows previous work on *self-paging*, which required each application to use its own resources (disk, memory and CPU) to deal with its own memory faults [5].

## 2.2 Memory virtualisation and sharing

A conventional operating system expects to own and manage a range of contiguously-addressed physical memory. Page tables in these systems translate *virtual addresses* into *physical addresses*. Since virtualisation can multiplex several guest operating systems on a single host, not all guests will receive such a range of physical memory. Furthermore, to ensure isolation, the VMM's and guests' memory must be protected from each other, so the VMM must ensure that all updates to the hardware page tables are valid.

Therefore, a virtualised system typically has three classes of address. *Virtual addresses* are the same as in a conventional OS. Each VM has a *pseudo-physical address space*, which is contiguous and starts at address zero. Finally, *machine addresses* refer to the physical location of memory in hardware. A common arrangement is for guests to maintain page tables that translate from virtual to pseudo-physical addresses, and the VMM to maintain separate *shadow page tables* that translate directly from virtual addresses to machine addresses [23]. A more recent approach is to use additional hardware to perform the translation from pseudo-physical addresses to machine addresses [10, 19]. Finally, it is also possible to modify the OS to use machine addresses and communicate with the VMM to update the hardware page tables explicitly [1].

Pseudo-physical addresses provide an additional layer of indirection that makes it possible to share memory between virtual machines. Since, for each VM, there is a pseudo-physical-to-machine (P2M) mapping, it is possible to make several *pseudo-physical frame numbers* (PFNs) map onto a single *machine frame number* (MFN). Therefore, if two VMs each have a page with the same contents, the VMM can update the P2M mapping and the shadow page tables to make those pages use the same machine frame. We discuss how other systems detect duplicates in Section 3, and the Satori approach in Section 4.

If two VMs share a page, an attempt to write to it must cause a page fault. This is achieved by marking the page read-only in the shadow page table. Such a page fault is called a *copy-on-write fault*. When this occurs, the VMM handles the fault by allocating a new frame and making a private copy of the page for the faulting guest. It also updates the P2M mapping and shadow page tables to ensure that the guest now uses the private copy.

A consequence of page sharing is that the memory used by a VM can both dynamically decrease (when a sharing opportunity is exploited) and dynamically increase (when sharing is broken). This presents

a resource allocation problem for the VMM. A conventional operating system does not have fine-grained, high-frequency mechanisms to deal with memory being added or removed at run time (*Memory hotplug* interfaces are unsuitable for frequent, page-granularity addition and removal [13]). Therefore, one option is to use a *balloon driver* in each guest, which pins physical memory within a guest and donates it back to the VMM [1, 23]. The "balloon" can inflate and deflate, which respectively decreases and increases the amount of physical memory available to a given VM.

However, a balloon driver requires cooperation from the guest: an alternative is *host paging*, whereby the VMM performs page replacement on guests' pseudo-physical memory [4, 23]. Host paging is expensive, because a VM must be paused while evicted pages are faulted in, and even if the VMM-level and OS-level page replacement policies are perfectly aligned, double paging (where an unused page must be paged in by the VMM when the OS decides to page it out) negatively affects performance. We deliberately avoid using host paging, and use a combination of the balloon driver (see § 4.2) and volatile pages (see § 4.3) to vary the memory in each guest dynamically.

*Collaborative memory management* (CMM) attempts to address the issue of double paging [14]. This system was implemented for Linux running on IBM's z/VM hypervisor for the zSeries architecture. In CMM, the guest VM provides hints to the VMM that suggest what pages are being used, and what pages may be evicted with little penalty. In Satori, we use part of this work for a different purpose: instead of using hints to improve a host pager, we use them to specify pages which may be reclaimed when sharing is broken (see § 5.3).

## 2.3 Enlightenment

Our approach to memory sharing is based on *enlightenments*, which involve making modifications to operating systems in order to achieve the best performance in a virtualised environment; in this paper we use the terms "enlightenment" and "paravirtualisation" interchangeably. Operating systems have been modified to run on VMMs for almost as long as VMMs have existed: the seminal VM/370 operating system employs *handshaking* to allow guests to communicate with the VMM for efficiency reasons [15]. "Paravirtualisation" was coined for the Denali VMM [25], and Xen was the first VMM to run paravirtualised commodity operating systems, such as Linux, BSD and Windows [1]. Xen showed that by paravirtualising the network and block devices, rather than relying on emulated physical hardware, it was possible to achieve near-native I/O speeds.

More extreme paravirtualisation has also been proposed. For example, Pfaff *et al.* designed Ventana as a *virtualisation-aware file system*, to replace virtual block devices as the storage primitive for one or more VMs [12]. This design concentrates on adding functionality to the file system—for example versioning, isolation and encapsulation—and considers sharing from the point of view of files shared between users. It does not specifically address resource management or aim to improve performance. Our approach is orthogonal to Ventana, and similar memory sharing benefits could be achieved with a virtualisation-aware file system. Indeed, using a system like Ventana would probably make it easier to identify candidates for sharing, and improve the overall efficiency of our approach.

Other systems, such as VMware ESX Server [23] and the Difference Engine [4] have a design goal of supporting unmodified guest OSs. In contrast, we have concentrated on paravirtualised guests for two reasons. First, there is an increasing trend towards enlightenments in both Linux and Microsoft Windows operating systems [18, 20]. Secondly, we believe that where there is a compelling performance benefit in using enlightenments, the necessary modifications will filter down into the vanilla releases of these OSs.

# 3 Related Work

Waldspurger described a broad range of memory management techniques employed in the VMware ESX Server hypervisor, including page sharing [23]. In VMware ESX Server, page sharing opportunities are discovered by periodically scanning the physical memory of each guest VM, and recording fingerprints of each page. When the scanner observes a repeated fingerprint, it compares the contents of the relevant two pages, and shares them if they are identical. In the same work, Waldspurger introduced the balloon driver that is used to alter guest memory allocations. However, since VMware ESX Server is designed to run unmodified guest operating systems, it must also support host paging. In Satori, we avoid host paging because of its negative performance impact (see § 2.2), and avoid memory scanning because it does not detect short-lived sharing opportunities (see § 4.1).

A contemporary research project has added page sharing to the Xen Virtual Machine Monitor. Vrable *et al.* began this effort with Potemkin [22], which uses *flash cloning* and *delta virtualization* to enable a large number of mostly-identical VMs on the same host. Flash cloning creates a new VM by copying an existing reference VM image, while delta virtualization provides copy-on-write sharing of memory between the original image and the new VM. Kloster *et al.* later extended this work with a memory scanner, similar to that found in VMware ESX Server [7]. Finally, Gupta *et al.* implemented the Difference Engine, which uses patching and compression to

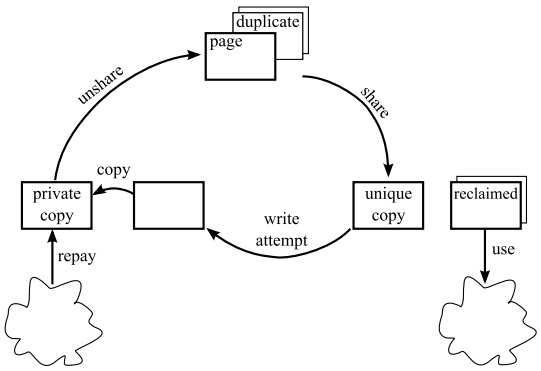



Figure 1: Sharing cycle

achieve greater memory savings than sharing alone. We have implemented Satori on Xen in a parallel effort, but we use guest OS enlightenments to reduce the cost of duplicate detection and memory balancing.

The Disco VMM includes some work on *transparent page sharing* [2]. In Disco, reading from a special *copy-on-write disk* involves checking to see if the same block is already present in main memory and, if so, creating a shared mapping to the existing page. We apply a similar policy for duplicate detection, as described in § 4.1. However, we also implement content-based sharing for disk I/O (§ 5.2), so it is not necessary to use copy-on-write disks, and furthermore we can exploit identical blocks within the same disk.

# 4 Design decisions

In this section, we present the major design decisions that differentiate Satori from previous work on page sharing [23, 4]. Figure 1 shows the life-cycle of a page that participates in sharing. This diagram raises three key questions, which we address in this section:

**How are duplicates detected?** We use *sharing-aware block devices* as a low-overhead mechanism for detecting duplicate pages. Since a large amount of sharing originates within the page cache, we monitor data as it enters the cache (§ 4.1).

**How are memory savings distributed?** When $n$ identical pages are discovered, these can be represented by a single physical page, and $n-1$ pages are saved. We distribute these savings to guest VMs in proportion with their contribution towards sharing (§ 4.2).

**What if sharing is broken?** Shared pages are necessarily read-only. When a guest VM attempts to write to a shared page, the hypervisor makes a writable private copy of the page for the guest. We require that the guest itself provides a list of *volatile* pages that may be used to provide the necessary memory for private copies. In addition, we obviate the need for copying in certain cases (§ 4.3).

We have taken care to ensure that our answers to the above questions are hypervisor-agnostic and may be implemented together or individually. Although our prototype uses the Xen VMM (see Section 5), these techniques should also be useful for developers of other hypervisors. In particular, our duplicate detection and savings distribution policies could be implemented without modifying core OS components in the guest VMs. However, by enlightening the guest OS, it is possible to achieve better performance, and we contend that our techniques are best implemented as a whole.

## 4.1 How are duplicates detected?

In order to exploit page sharing, it is necessary to detect duplicate pages. As described in § 2.2, the most common approach to this problem is to scan the memory of all guest VMs periodically, and build up a list of page fingerprints that can be used to compare page contents. In this subsection, we propose *sharing-aware block devices* as a more efficient alternative. We discuss the problems with the scanning-based approach, and explain why the block interface is an appropriate point at which to detect potential sharing.

As we show in § 6.1, many sharing opportunities are short-lived, and yet these provide a large amount of overall sharing when taken as a whole. In principle, the memory scanning algorithm is exhaustive and all duplicates will eventually be found. However, in practise the rate of scanning has to be capped: in the extreme case, each memory write would trigger fingerprint recomputation. For example in VMware ESX Server the default memory scan frequency is set to once an hour, with a maximum of six times per hour [21]. Therefore, the theoretical mean *duplicate discovery time* for the default setting is 40min, which means that short-lived sharing opportunities will be missed. (We note that there are at least three relevant configuration options: the scan period, scan throughput (in MB per second per GHz of CPU), and maximum scan rate (in pages per second). In our evaluation (§ 6.1), the "aggressive" settings for VMware use the maximum for all three of these parameters.)

When an operating system loads data from disk, it is stored in the page cache, and other researchers have noted that between $63.8\%$ and $93.0\%$ of shareable pages between VMs are part of the page cache [8]. For example, VMs based on the same operating system will load identical program binaries, configuration files and data files. (In these systems, the kernel text will also be identical, but this is loaded by Xen domain builder (bootloader), and does not appear in the page cache. Though we do not implement it here, we could modify the Xen domain builder to provide sharing hints.)

The efficacy of sharing-aware block devices relies

on the observation that, for the purpose of detecting duplicates, a good image of the page cache contents can be built up by observing the content of disk reads. While this approach does not capture any subsequent in-memory writes, we do not expect the sharing potential of dirty pages to be high. Since a VMM uses virtual devices to represent block devices, we have a convenient place to intercept block reads. We describe our implementation of sharing-aware block devices in § 5.2.

The situation improves further if several guests share a common base image for their block device. When deploying a virtualised system, it is common to use a single *substrate* disk for several VMs, and store VM-private modifications in a copy-on-write overlay file. If a guest reads a block from the read-only substrate disk, the block number is sufficient to identify it uniquely, and there is no need to inspect its contents. This scheme has the additional advantage that some reads can be satisfied without accessing the underlying physical device.

Previous work on page sharing emphasises zero pages as a large source of page duplicates. Clearly, these pages would not be found by block-device interposition. However, we take a critical view of zero-page sharing. An abundance of zero pages is often indicative of low memory utilisation, especially in operating systems which implement a scrubber. We believe that free page sharing is usually counterproductive, because it gives a false sense of memory availability. Consider the example of a lightly loaded VM, in which $50\%$ of pages are zero pages. If these pages are reclaimed and used to run another VM, the original VM will effectively run with $50\%$ of its initial allocation. If this is insufficient to handle subsequent memory demands, the second VM will have to relinquish its resources. We believe that free memory balancing should be explicit: a guest with low memory utilisation should have its allocation decreased. Several systems that perform this resource management automatically have been proposed [26].

## 4.2 How are memory savings distributed?

The objective of any memory sharing mechanism is to reuse the reclaimed pages in order to pay for the cost of running the sharing machinery. A common approach is to add the extra memory to a global pool, which can be used to create additional VMs [4]. However, we believe that only the VMs that participate in sharing should reap the benefits of additional memory. This creates an incentive for VMs to share memory, and prevents malicious VMs from negatively affecting the performance of other VMs on the same host. Therefore, Satori distributes reclaimed memory in proportion to the amount of memory that each VM shares.

When Satori identifies $n$ duplicates of the same page, it will reclaim $n-1$ redundant copies. In the common case of $n = 2$, our policy awards each of the contributing VMs with an entitlement of $0.5$ pages—or, more generally, $\frac{n-1}{n}$ pages—for *each* shared page (Figure 2). For each page of physical memory, $p$, we define $n(p) \in \mathbb{N}$ as the *sharing rank* of that page. For VM $i$, which uses the set of pages $M(i)$, the total *sharing entitlement*, $s(i)$, is calculated as follows:

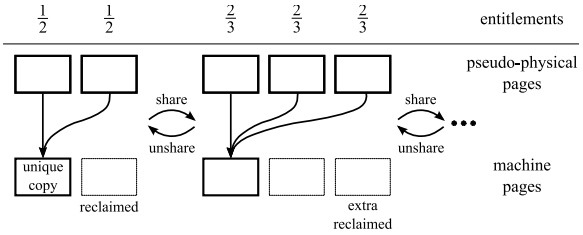



Figure 2: Sharing entitlement calculation

$$s(i) \quad = \quad \sum_{p \in M(i)} \frac{n(p) - 1}{n(p)}$$

Satori interrogates the sharing entitlements for each VM every second, and makes the appropriate amount of memory available to the VMs.

The sharing rank of a particular page will not necessarily remain constant through the lifetime of the sharing, since additional duplicates may be found, and existing duplicates may be removed. Therefore, the sharing entitlement arising from that page may change. Consider what happens when a new duplicate is discovered for an already $n$-way shared page. The VM that provided the new duplicate will receive an entitlement of $\frac{n}{n+1}$ pages, and the owners of the existing $n$ duplicates will see their entitlement increase by $\frac{n}{n+1} - \frac{n-1}{n} = \frac{1}{n(n+1)}$ for each copy they own. Similarly, the entitlements must be adjusted when a shared page departs.

In Satori, guests claim their sharing entitlement using *memory balloons* [23]. When the entitlement increases, the balloon deflates and releases additional pages to the guest kernel. In our implementation we set up the guests to always claim memory as soon as it becomes available. However, guests can elect to use more complex policies. For example a guest may refrain from using its entitlement if it experiences low memory demand, or expects its shared pages to be short-lived. We have explicitly avoided using host paging to deal with fluctuating memory allocations. As a result, our implementation is simpler, and we have avoided the well-known problems associated with host paging. However, without host paging, we have to guarantee that the hypervisor can recover memory from the guests when it needs to create private copies of previously-shared pages. In the next subsection, we introduce the *repayment FIFO*, which addresses this issue.

### 4.3 What if sharing is broken?

If two or more VMs share the same copy of a page, and one VM attempts to write to it, the VMM makes a private copy of the page. Where does the VMM get memory for this copy?

Satori obtains this memory from a guest-maintained *repayment FIFO*, which contains a list of pages the guest is willing to give up *without* prior notification. The size of a VM's repayment FIFO must be greater than or equal to its sharing entitlement. Our approach has three major advantages: (a) the hypervisor can obtain pages quickly, as there is no synchronous involvement with the guest, (b) there is no need for host paging, and (c) there is no risk that guest will be unable to relinquish resources due to double copy-on-write faults (i.e. a fault in the copy-on-write fault handler).

Pages in the repayment FIFO must not contain any irreplaceable information, because the guest will not have a chance to save their contents before the hypervisor reclaims them. Memory management subsystems already maintain book-keeping information about each page, which makes it possible to nominate such *volatile pages* without invasive changes.

In Satori the hypervisor uses sharing entitlements to determine the VM from which to reclaim memory. It does so by inspecting how much memory each VM drew from the sharing mechanism, in comparison to its current sharing entitlement. Since the sum of sharing entitlements is guaranteed to be smaller or equal to the number of removed duplicate pages, there will always be at least one VM with a negative memory balance (i.e. the VM drew more than its entitlement). Note that only the VMs which are involved in the broken sharing will be affected. This is essential to maintain performance isolation, as a malicious VM will be unable to affect any VMs with which it does not share memory.

A special case of broken sharing is when a page is reallocated for another purpose. For example, a guest may decide to evict a shared page from the page cache, scrub its content and reallocate it. In a copy-on-write system, the scrubber would cause a page fault when it begins to scrub the page, and the VMM would wastefully copy the old contents to produce a private version of the page. We use a scheme called *no-copy-on-write*, which informs the VMM that a page is being reallocated, and instead allocates a zero page (from a pre-scrubbed pool) for the private version.

To the best of our knowledge, Satori is the first system to address a covert channel created by memory sharing. An attacker can infer the contents of a page in another guest, by inducing sharing with that page and measuring the amount of time it takes to complete a write. (If a page has been shared, the processing of a copy-on-write fault will measurably increase the write latency.) For example, we might want to protect the identity of server processes running in a guest, because security vulnerabilities might later be found in them. We allow guests to protect sensitive data by specifying which pages should never be shared. Any attempts to share with these pages will be ignored by the hypervisor.

## 5 Implementation

We implemented Satori for Xen version 3.1 and Linux version 2.6.18 in 11551 lines of code (5351 in the Xen hypervisor, 3894 in the Xen tools and 2306 in Linux). We chose Xen because it has extensive support for paravirtualised guests [1]. In this section, we describe how we implemented the design decisions from Section 4.

Our changes can be broken down into three main categories. We first modified the Xen hypervisor, in order to add support for sharing pages between VMs (§ 5.1). Next, we added support for sharing-aware block devices to the Xen control tools (§ 5.2). Finally, we enlightened the guest operating system, so that it can take advantage of additional memory and repay that memory when necessary (§ 5.3).

### 5.1 Hypervisor modifications

The majority of our changes were contained in the hypervisor. First of all, the upstream version of Xen does not support transparent page sharing between VMs, so it was necessary to modify the memory management subsystem. Once this support was in place, we added a hypercall interface that the control tools use to inform the hypervisor that pages may potentially be shared. Finally, we modified the page fault handler to deal with instances of broken sharing.

In § 2.2, we explained that each VM has a contiguous, zero-based pseudo-physical address space, and a P2M mapping for converting pseudo-physical addresses to machine addresses. To support transparent page sharing, it is necessary to allow multiple pseudo-physical pages to map to a single frame of machine memory. Furthermore, the machine frame that backs a given pseudo-physical page may change due to sharing. Therefore, it is simplest to use shadow page tables in the guest VMs. However, regular paravirtualised guests in Xen do not use shadow page tables, so we ported this feature from the code which supports fully-virtualised guests. In addition, we had to modify the reference counting mechanism used in Xen to keep track of page owners. In Xen each page has a single owner, so we added a synthetic "sharing domain" which owns all shared pages.

As described in § 5.3, we maintain information about the state of each (pseudo-)physical page in each guest. Both the guest and the hypervisor may update this information, so it is held in a structure that is shared between the hypervisor and the guest. The hypervisor uses this

structure to select which page should be used to satisfy a copy-on-write fault (either a page from the repayment FIFO, or, in the no-copy-on-write case, a zero-page).

We export the sharing functionality to the guest through the hypercall interface. We add three new hypercalls, named `share_mfns`, `mark_ro` and `get_ro_ref`.

The `share_mfns` hypercall takes two machine frame numbers (MFNs)—a source MFN and a client MFN—and informs the hypervisor that all pseudo-physical pages backed by the client frame should now use the source frame. The hypercall works as follows:

1. Mark the source and client frame as read-only, if they are not already.
2. Compare the contents of the source and client frame. If they are not equal, return an error.
3. Remove all mappings to the client MFN from the shadow page tables.
4. Update the relevant P2M mappings to indicate that the source frame should be used in place of the client frame.
5. Free the client frame for use by the guest VMs.

Note that this hypercall is not guaranteed to succeed. For example, after the duplicate detector notices that two pages are the same, but before they are marked read only, a guest might change the contents of one of the pages. Therefore, the hypercall may fail, but there is no risk that the contents of memory will be incorrect: the source and client frame will continue to be used as before.

For copy-on-write disks, we want to make an early decision about whether or not physical I/O will be required. Therefore, we use the `mark_ro` hypercall to enforce read-only status on all pages that are read from the read-only substrate. (Technically, we make a page read-only by treating it as 1-way shared; if the guest writes to it, the sharing is simply broken by marking the page as writable and changing the owner to the guest.) The complementary `get_ro_ref` hypercall ensures that the contents of the frame have not been changed (i.e. that the MFN is still read-only), and increments the sharing reference count to prevent it from being discarded. We describe the copy-on-write disk support in § 5.2.

The final hypervisor component required for page sharing is a modified page fault handler. We added two new types of page fault, which Xen must handle differently. The first is a straightforward *copy-on-write fault*, which is triggered when a guest attempts to write to a shared page. In this case, the handler recalculates the sharing entitlements for the affected guests, and reclaims a page from one of the guests that now has claimed more memory than its entitlement. The handler removes this page from the appropriate guest's repayment FIFO and copies in the contents of the faulting page. We also add a *discard fault*, which arises when a guest attempts to access a previously-volatile page that the VMM has reclaimed. If so, the handler injects this fault into the guest, as described in § 5.3.

## 5.2 Sharing-aware block devices

We implemented duplicate detection using *sharing-aware block devices*. Xen provides a high-performance, flexible interface for block I/O using *split devices*. The guest contains a *front-end driver*, which presents itself to the guest OS as a regular block device, while the control VM hosts a corresponding *back-end driver*. Previous work has shown how the back-end is a suitable interposition point for various applications [24], in particular for creating a distributed storage system [9]. We use the existing *block-tap* architecture to add duplicate detection.

The key steps in a block-tap read request are as follows:

1. The front-end (in the guest) issues a read request to the back-end through the inter-VM *device channel*, by providing a block number and an I/O buffer.
2. The back-end maps the I/O buffer into a user-space control tool, called `tapdisk`.
3. `tapdisk` performs device-specific processing for the given block number, and returns control to the back-end driver.
4. The back-end unmaps the I/O buffer and notifies the front-end of completion.

Since `tapdisk` is implemented as a user-space process and provides access to I/O data, it is simple to add custom block-handling code at this point. Satori modifies the `tapdisk` read path in order to record information about what data is loaded into which locations in the guests' memory. We developed two versions of duplicate detection: content-based sharing, and copy-on-write disk sharing.

For content-based sharing, we hash the contents of each block as it is read from disk. We use the hash as the key in a hashtable, which stores mappings from hash values to machine frame numbers (MFNs). First, we look for a match in this table, and, if this is successful, the resulting MFN is a candidate for sharing with the I/O buffer. Note that the MFN is merely a hint: the contents of that frame could have changed, but since we have already loaded the data into the I/O buffer, it is acceptable for the sharing attempt to fail. If the hash is not present in the hashtable, we invalidate any previous entry that maps to the I/O buffer's MFN, and store the new hash-to-MFN mapping.

For copy-on-write disk sharing, the process is slightly different (see Figure 3). The first time a block is read from the substrate disk, Satori invokes the `mark_ro` hypercall on that page, and stores a mapping from the

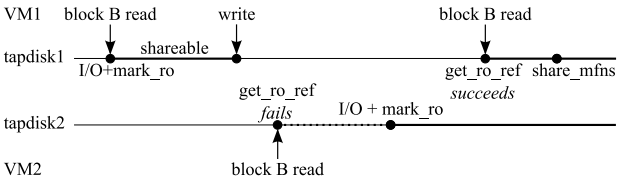



Figure 3: `mark_ro` and `get_ro_ref` usage for copy-on-write disks.

*block number* to the I/O buffer MFN. (If the guest subsequently writes to the page before it is shared, the read-only status is removed.) On subsequent reads, Satori consults the block number-to-MFN mapping to see if the block is already cached in memory. If it is, Satori invokes the `get_ro_ref` hypercall on the MFN, which, if it succeeds, ensures that the subsequent call to `share_mfns` will be successful. If the block number is not found in the mapping table or if `get_ro_ref` fails, Satori must request physical disk I/O to read the appropriate block. At this point, a second look-up could be used to detect content-based sharing opportunities.

The Xen architecture places each virtual block device in a separate `tapdisk` process, to simplify management and improve fault isolation. However, we need to share information between devices, so it was necessary to add a single additional process, called `spcctrl` (**S**hared **P**age **C**ache **C**on**TR**o**L**ler), which hosts the mappings between content hashes or block numbers, and MFNs. The `tapdisk` processes communicate with `spcctrl` using pipes, and our current implementation of `spcctrl` is single-threaded.

## 5.3 Guest enlightenments

In Satori, we have used enlightenments to obtain OS-level information about guest pages. These extend the existing paravirtualised Linux guests, which Xen already supports [1].

For Satori, the most important enlightenment is adding the repayment FIFO to the guest kernel. Recall that the repayment FIFO is a list of *volatile pages*, i.e. physical pages that the operating system is willing to relinquish at any time (in particular, when sharing is broken and a new frame is needed for a private copy). Since the guest must relinquish these pages without warning, it is essential that their contents can be reconstructed from another source. Hence an obvious source of volatile pages is the set of clean pages in the page cache. We paravirtualised the Linux page cache to provide *page hints* about volatile pages.

We based our implementation of volatile pages on earlier work on Collaborative Memory Management (CMM) [14]. CMM is, in essence, a memory controller which relies on page states (especially page volatility) to dynamically adjust the available (machine) memory for each guest. CMM is implemented for the IBM zSeries z/VM hypervisor, but the majority of the code is architecture-independent, as it deals with page-state transitions in the Linux page and swap caches. We built on CMM's page hinting by adding support for the x86 architecture and the Xen hypervisor.

The major difference between x86 and zSeries (s390) in the context of volatile pages is the handling of dirtying. The x86 architecture maintains dirty bits for virtual pages in the PTE, whereas the s390 architecture maintains a dirty bit for each machine page. Since a given page can only become volatile if it is not dirty, we implemented a machine-page-level dirty bit in software for the x86 architecture. Our approach is more conservative than is strictly necessary, because we consider the existence of any writable mapping to dirty the page, even if there was no actual write.

Satori uses a shared structure between Xen and each guest to store and modify page states (as discussed in § 5.1). The page states read from this structure are used in the guest page fault handler to distinguish between "regular" and *discard* faults. On a discard fault, Linux uses reverse mappings to remove all references to the discarded page, effectively removing the page from its respective cache (page or swap).

We also use the instrumentation in the page allocator, already present in order to drive page state transitions, to support the no-copy-on-write policy. Whenever a page is reallocated, we update the shared page state structure to reflect this. On a write fault to a shared page, Xen checks to see whether the page has been reallocated, and, if so, provides a page from its zero page cache.

In addition, we have added support to guests for specifying that some pages must not be shared (to avoid the secret-stealing attack described in § 4.3). At present, we allow the guest to specify that a set of pseudo-physical pages must never be shared (i.e. all calls to `share_mfns` or `get_ro_ref` will fail).

# 6 Evaluation

To characterise Satori's performance, we have conducted an evaluation in three parts. First, we have profiled the opportunities for page sharing under different workloads (§ 6.1). In contrast with previous work, we specifically consider the *duration* of each sharing opportunity, as this is crucial to the utility of page sharing. We then measure the effectiveness of Satori, and show that it is capable of quickly detecting a large amount of sharing (§ 6.2). Finally, we measure the effect that Satori has on performance, in terms of the benefit when sharing is enabled, and the overhead on I/O operations (§ 6.3).

For our tests we used two Dell PowerEdge 1425 servers each equipped with two 2.8 GHz Xeon CPUs, 2.5 GB of RAM and an 80 GB Seagate SATA disk. VMs

| Rank | Pages saved | Percentage saving |
|---|---|---|
| 2 | 1565421 | 79.7% |
| 3 | 137712 | 7.01% |
| 4 | 59790 | 3.04% |
| 5 | 18760 | 0.96% |
| 6 | 24850 | 1.27% |
| 8 | 10059 | 0.51% |
| 10 | 10467 | 0.53% |
| 14 | 10218 | 0.52% |
| others | 126865 | 6.46% |

Table 1: Breakdown of sharing opportunities by rank (excluding zero-pages).

ran Ubuntu Linux 8.04 in all cases, except for two experiments, for which we state the OS version explicitly.

In the following subsections, we make repeated reference to several workloads, which we abbreviate as follows:

- KBUILD-256 Vanilla Linux 2.6.24 kernel build with 256 MB of physical memory.
- KBUILD-512 As KBUILD-256, with 512 MB.
- HTTPERF httperf benchmark [6] run against Apache web-server with 512 MB of memory, serving randomly generated static webpages.
- RUBIS RUBiS web auction application with 512 MB, serving requests generated by the default client workload generator [11].

## 6.1 Sharing opportunities

The major difference between Satori and contemporary page sharing schemes is that it can share many identical pages as soon as they are populated. In this subsection, we show that a substantial proportion of sharing is short-lived. Therefore, Satori is much more likely to exploit this sharing than schemes that rely on periodically *scanning* physical memory, looking for identical page contents [23, 4].

To analyse the sharing opportunities, we ran each of the KBUILD-256, KBUILD-512, HTTPERF and RUBIS workloads in two virtual machines for 30 minutes, and took a memory dump every 30 seconds.

Table 1 shows the number of pages that can be freed using page sharing, for each rank. (In a sharing of rank $n$, $n$ identical pages map to a single physical page.) The figures are an aggregate, based on the total of 60 memory dumps sampled from pairs of VMs running the KBUILD-512, HTTPERF and RUBIS workloads. Note that most sharing opportunities have rank 2: i.e. two identical pages exist and can be combined into a single physical page.

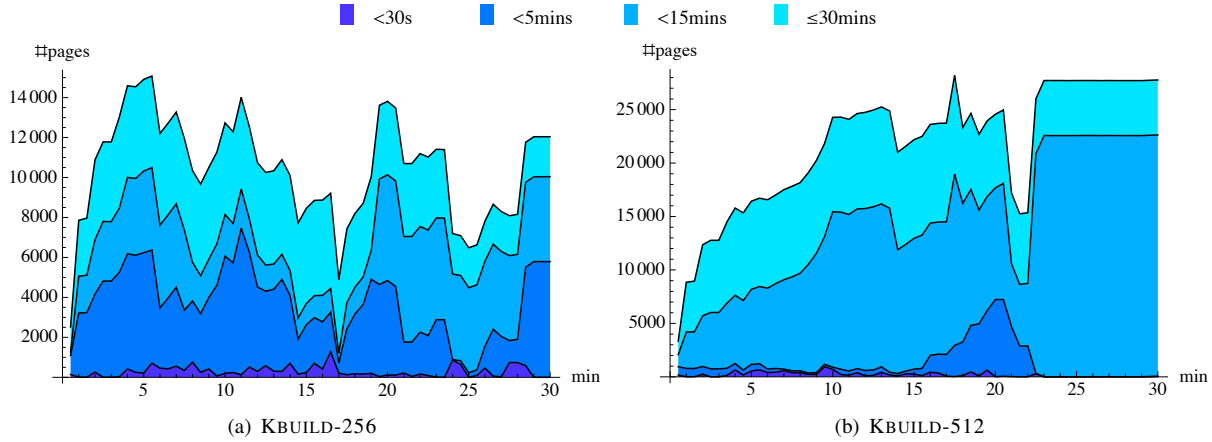
| Operation | Count | Total (ms) | Avg ($\mu$s) |
|---|---|---|---|
| `mark_ro` | 127479 | 5634 | 44.1 |
| `share_mfns` | 61905 | 474 | 7.7 |
| `get_ro_ref` | 69124 | 64 | 0.9 |
| Total | 258508 | 6172 | — |

Table 2: Breakdown of Satori hypercalls during HTTPERF workload

Figure 4 compares the number of *unique* shared pages during the KBUILD-256 and KBUILD-512 workloads. (By considering only unique shared pages, we underestimate the amount of savings for pages with rank $> 2$. Table 1 demonstrates that the majority of shareable pages have rank 2, except zero pages, which we address separately below.) We have divided the sharing opportunities into four duration ranges. The figures clearly show that a substantial amount of sharing is short-lived, especially in a more memory-constrained setup (KBUILD-256). Also, the amount of sharing for the KBUILD-512 workload is approximately twice as much as that for KBUILD-256, because of less contention in the page cache. Finally, the kernel build process completes 6 minutes sooner with 512 MB of memory: this makes the benefits of additional memory clear.

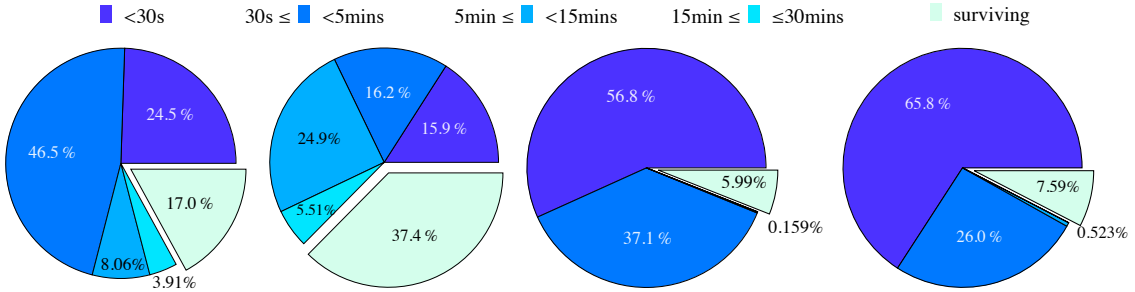
Figure 5 separately categorises shareable non-zero pages and zero pages into the same duration ranges as Figure 4. It should be noted that the number of sharing opportunities arising from zero pages (Figures 5(c) and 5(d)) is approximately 20 times greater than from non-zero pages (Figures 5(a) and 5(b)). However, more than 90% of zero-page sharing opportunities exist for less than five minutes. This supports our argument that the benefits of zero-page sharing are illusory.

In § 4.1, we stated that, on average, it will take 40 minutes for VMware ESX Server to detect a duplicate page using its default page scanning options. We ran the following experiment to validate this claim. Two VMs ran a process which read the same 256 MB, randomly-generated file into memory, and Figure 6 shows the number of shared pages as time progresses. The lower curve, representing the default settings, shows that half of the file contents are shared after 37 minutes, which is close to our predicted value; the acceleration is likely due to undocumented optimisations in VMware ESX Server. The higher curve shows the results of the same experiment when using the most aggressive scanning options. Using the same analysis, we would expect a duplicate on average to be detected after 7 minutes. In our experiment, half the pages were detected after almost 20 minutes, and we suspect that this is a result of the aggressive settings causing the page hint cache to be flushed more often.




(a) KBUILD-256 (b) KBUILD-512

Figure 4: Sharing opportunities during the execution of workloads KBUILD-256 and KBUILD-512.




(a) KBUILD-256 (b) KBUILD-512 (c) KBUILD-256 (d) KBUILD-512

Figure 5: Duration of page sharing opportunities for kernel compilation workloads. (a) and (b) show non-zero pages, (c) and (d) zero pages. The exploded sectors show sharings left at the end of the experiment.

## 6.2 Satori effectiveness

In the next set of experiments, we measured the amount of sharing that Satori achieved using sharing-aware block devices. We also examined how the surplus memory was distributed between individual virtual machines.

The first experiment used two pairs of virtual machines. Two VMs each ran the HTTPERF-256 workload, i.e. the HTTPERF workload with 256 MB of memory (rather than 512 MB). Because the aggregate amount of memory was insufficient to cache the entire data set in memory, the number of shareable pages varied as data was loaded into and evicted from each VM's page cache. The other two VMs each ran the KBUILD-512 workload; however they used Debian Linux rather than Ubuntu.

Figure 7 shows that the sharing entitlements for the VMs running KBUILD-512 are unaffected by the highly variable amount of sharing between the two HTTPERF workloads. Also, because we used different OSes for each pair of VMs, the sharing entitlements achieved before the workloads started (5 to 6 minutes after the measurements began) differ by about 30%.

Next, we ran two instances of a workload in separate VMs for 30 minutes, and repeated the experiment for the KBUILD-256, KBUILD-512, HTTPERF and RUBIS workloads. We ran these experiments under Satori and measured the number of shared pages, and compared these to memory dumps using the same methodology as described in § 6.1.

Figure 8 summarises the amount of sharing that Satori achieves for each workload. Satori performs best with the HTTPERF workload, shown in Figure 8(c). In this case, it achieves 94% of the total sharing opportunities, which is to be expected since HTTPERF involves serving a large volume of static content over HTTP, and the majority of the data is read straight from disk. The RUBIS workload performs similarly, with Satori achieving 91% of the total. The kernel compilation workloads, KBUILD-256 and KBUILD-512, perform less well. KBUILD-512 achieves about 50% of the total sharing opportunities until the very end of the build, when the kernel image is assembled from individual object files. KBUILD-256 is more memory-constrained, which forces the OS to flush dirty (non-shareable) caches.

Finally, we ran two experiments which evaluated Satori in a more heterogeneous environment. In the

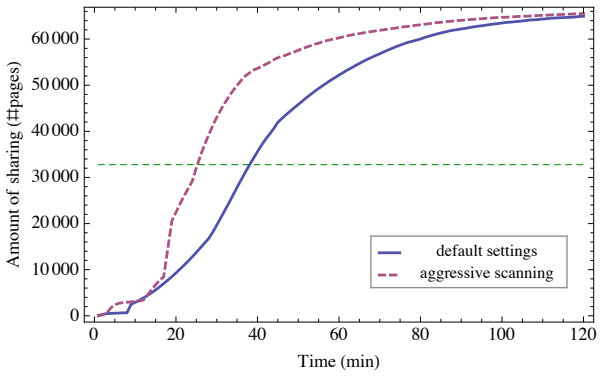



Figure 6: Sharing as time progresses for default and aggressive scanning settings in VMware ESX Server.

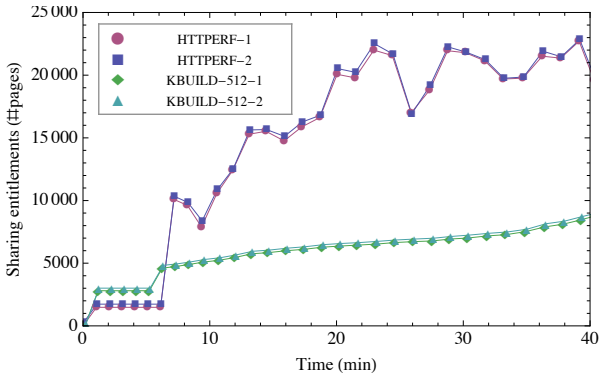



Figure 7: Sharing entitlements for two KBUILD-512 and two HTTPERF-256 workloads executing simultaneously.

first experiment, two VMs running the same version of Ubuntu Linux performed the HTTPERF and RUBiS workloads. In this setup Satori was able to exploit over 70% of the total sharing opportunities. (The remaining 30% was mostly due to the identical kernel images, which the current version of Satori does not detect.). In the second experiment, we used the same workloads with different guest OSs (Ubuntu and Debian respectively). In this setup, 11 MB of sharing was theoretically possible, and only because the two distributions use an identical kernel. In this case, Satori could only achieve approximately 1 MB of savings (9% of the total).

Although Satori achieves varying results in terms of memory savings, recall that these results come solely from using our enlightened block device. These results show that we can exploit up to 94% (for HTTPERF) of the total sharing opportunities through this method alone. The alternative approach, which involves scanning memory, incurs a constant overhead at run-time, and must be rate-limited to prevent performance degradation [21]. The Difference Engine exhibits an overhead of up to 7% on some macrobenchmarks, though this includes the overhead of page compression and sub-page-level patching [4]. Satori provides a flexible interface for adding other sharing policies: we are developing a tool that systemically identifies the source(s) of other sharing opportunities. We hope that this will lead to additional enlightenments that improve Satori's coverage.

In § 4.3, we described an attack on memory sharing that allows a VM to identify sensitive data in another guest. On VMware ESX Server, we were able to determine the precise version of `sshd` running in another guest, by loading a page from each of 50 common distribution-supplied `sshd` binaries into memory, and periodically measuring the write latency to these pages. (On our hardware, we observed a 28-times increase for the matching page.) In Satori, we were able to protect the entire `sshd` address space, and, as a result, this attack failed.

## 6.3 Performance impact

We initially stated that memory sharing is desirable because it can improve the overall performance of VMs running on the same physical machine. In this subsection, we investigate the performance characteristics of Satori under various workloads. First, we measure *negative* impact: Satori introduces new operations for sharing memory, and these incur a measurable cost. We measure the cost of each sharing-related hypercall, and the overall effect on disk I/O. However, we then show that, for realistic macrobenchmarks, the overhead is insignificant, and the additional memory can improve overall performance.

To measure the cost of individual sharing-related operations, we instrumented the Xen hypervisor to record the number and duration of each hypercall. Table 2 shows the results for a 30-minute HTTPERF workload. The first thing to note is that Satori-related operations account for less than 6.2 seconds of the 30-minute benchmark. Of the individual operations, `mark_ro` is the most expensive, as it must occasionally perform a brute-force search of the shadow page tables for all mappings of the page to be marked read-only. We could optimise performance in this case by making the guest VM exchange back-reference information with the hypervisor, but the overall improvement would be negligible.

Satori detects sharing by monitoring block-device reads, and therefore the majority of its overhead is felt when reading data from disk. In order to measure this overhead, and stress-test our implementation, we ran the Bonnie filesystem benchmark in a guest VM against a sharing-aware block device. Table 3 shows a breakdown of read bandwidths. We ran the benchmark in four configurations, and repeated each experiment five times. In the baseline configuration, we disabled all Satori mechanisms. In successive configurations, we enabled content hashing, IPC with `spcctrl`, and finally hash lookup,

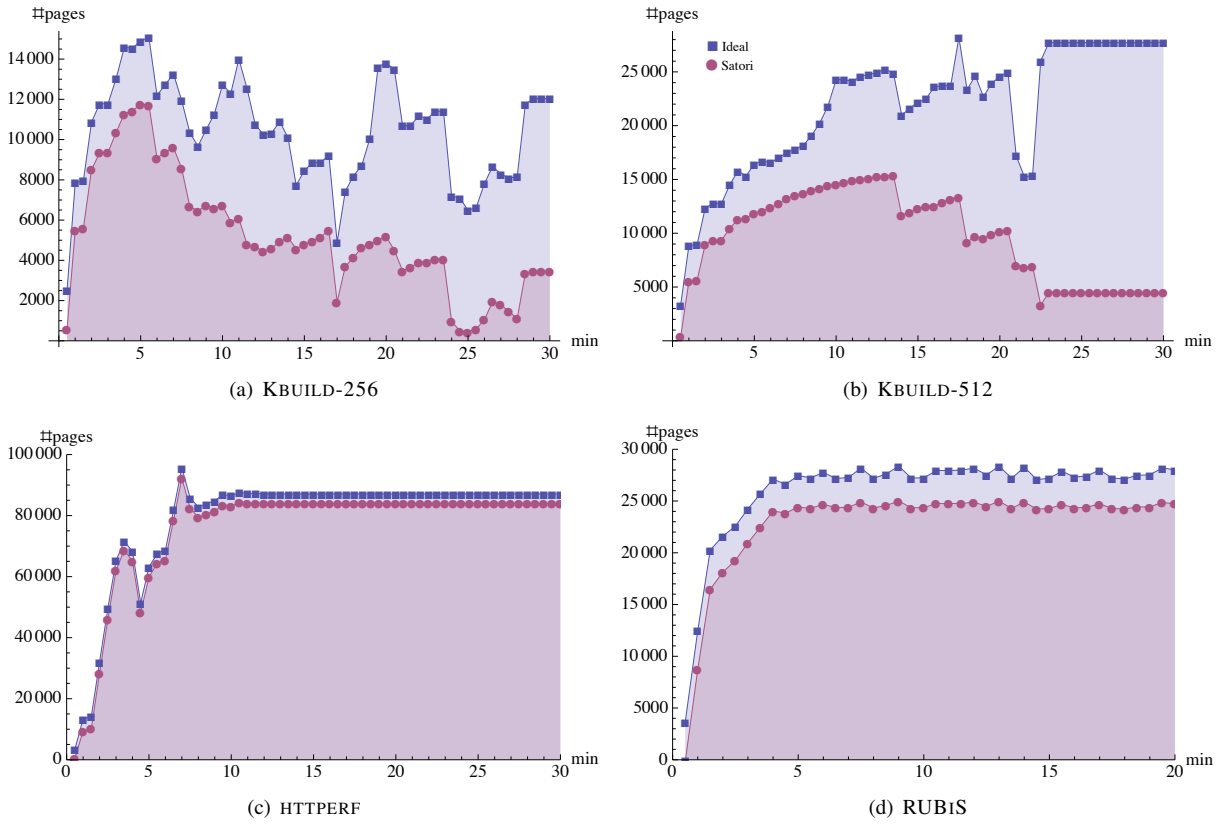



Figure 8: Amount of sharing achieved by Satori for each of the four main workloads (no zero-pages)

in order to isolate the performance impact of each function. Table 3 reports bandwidth figures for reads using `getc()`, and "intelligent reads", which use a block size of 16384 bytes.

The first thing to note is that Bonnie performs *sequential* reads on a 512 MB file, so the effect of any computation on the I/O path is amplified. (The impact of Satori on random reads is negligible.) Therefore, the 34.8% overhead for chunked reads with Satori fully enabled is a worst-case figure, and is to be expected. With a realistic workload, the I/O cost is likely to dominate. Nevertheless, it is instructive to consider the individual overheads:

- The overhead involved in hashing is relatively constant and less than 0.4%.
- IPC with the `spcctrl` process is costly. The present implementation uses UNIX pipes to communicate with `spcctrl`, which involves two additional context switches per read request. We plan to redesign this component to store the hashtable in a shared memory segment.
- The relative overhead of fully-enabled Satori is worse in the chunked read case, because less time is being wasted in making repeated system calls in the guest VM.

While we are continuing to improve Satori's performance, and eliminate these bottlenecks, we have only encountered the above issues when running Bonnie. For example, we ran a stripped-down kernel compilation in a single VM, which took an average of 780 seconds with Satori disabled, and 779 seconds with Satori fully enabled. Since the standard deviation over five runs was 27 seconds, it is clear that the overhead is statistically insignificant. In this experiment, the workload ran in isolation, and there were no benefits from sharing. As we will see next, the advantage of having additional memory can improve performance for many workloads.

We first ran an experiment to illustrate the benefit of memory sharing between VMs that share a copy-on-write disk. We ran a workload that read the contents of a copy-on-write disk into memory in a pseudorandom order. Five seconds later (while the first read was ongoing), we started the same workload, reading from the same disk in the same sequence, in another VM. Figure 9 shows the progress that both VMs achieved as a proportional gradient. VM1 reads at a consistent rate of 4.96 MB/s. When the workload commences in VM2, its initial rate is 111 MB/s, as the data that it reads can be provided by sharing memory with the page cache in VM1. After 0.22 seconds, VM2 has read all of the data held

| Mode | Read bandwidth (MB/s) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | `getc()` | | | | "Intelligent read" | | | |
| | Min | Max | Avg | Overhead | Min | Max | Avg | Overhead |
| No sharing | 26.9 | 28.2 | 27.6 | — | 47.1 | 47.4 | 47.4 | — |
| Hashing only | 26.1 | 28.4 | 27.5 | 0.4% | 47.1 | 47.4 | 47.3 | 0.2% |
| Hashing + IPC | 22.7 | 23.8 | 23.2 | 15.9% | 31.8 | 33.0 | 32.4 | 31.6% |
| Sharing enabled | 23.2 | 24.9 | 24.2 | 12.9% | 30.7 | 31.1 | 30.9 | 34.8% |

Table 3: Results of the Bonnie filesystem benchmark on Satori

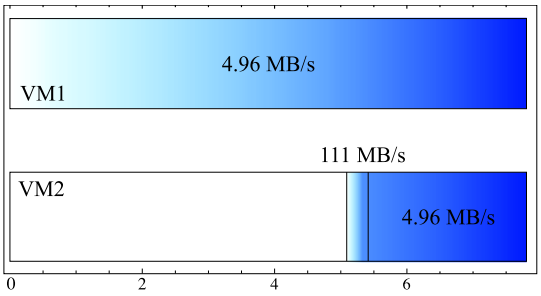



Figure 9: Copy-on-write disk read rates

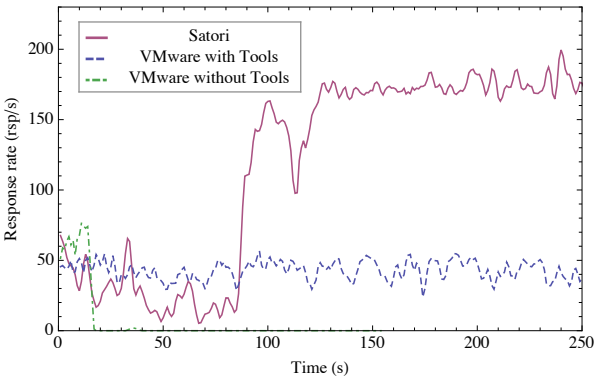



Figure 10: Aggregate HTTPERF response rates for the two VMs running on Satori, VMware, and VMware with VMware Tools

by VM1, and the two VMs become mutually synchronised, at the original rate, which is limited by the disk access time. Although this example is artificial, it shows Satori's effectiveness at exploiting page cache sharing for copy-on-write disks. Many recent cloud computing systems, such as Amazon's EC2 [17], encourage the use of standard *machine image* files, which are natural candidates for a copy-on-write implementation. Satori would be particularly effective in this case.

Finally, we ran the HTTPERF workload in two VMs as a macrobenchmark, to discover how well Satori exploits the extra memory that is made available through sharing. We compare Satori to VMware ESX Server—the leading commercial hypervisor—which uses the techniques described by Waldspurger to achieve page sharing and memory overcommitment [23].

Figure 10 shows how the aggregate HTTPERF response rate changes over time for Satori and VMware (with and without VMware Tools). The performance of Satori can be divided into two phases. First, it achieves approximately 30 responses per second while the cache is being loaded, which takes approximately 85 seconds. The response rate then jumps to between 170 and 200 responses per second as all subsequent requests can be satisfied from caches. In order to maintain these response rates, the VMs use their sharing entitlements to increase their page cache sizes. The physical memory available to each VM grows to over 770 MB over the first 120 seconds of the experiment.

The results for VMware are interesting. We note first that it was necessary to install the VMware Tools (which include a balloon driver) in order to achieve performance that was comparable to Satori. Without the VMware Tools, the VMM begins paging after approximately 15 seconds, and throughput drops almost to zero. Once host paging starts, the throughput only recovers occasionally, and never to more than 5 responses per second. With the VMware Tools installed, we observed that balloon permanently limited each VMs physical memory allocation to 500 MB. Therefore, the VMs were able to make progress without host paging, but the data set did not fit in the cache, and the response rate remained at around 40 responses per second. VMware was unable to establish sufficient sharing because the lifetime of a page in either page cache was usually too short for the memory scanner to find it.

## 7 Conclusions

We described Satori, which employs enlightenments to improve the effectiveness and efficiency of page sharing in virtualised environments. We have identified several cases where the traditional page sharing approach (i.e. periodic memory scanning) does not discover or exploit opportunities for sharing. We have shown that, by using information from the guest VMs, and making small modifications to the operating systems, it is possible to discover a large fraction of the sharing opportunities with insignificant overhead.

Our implementation has concentrated on sharing-aware block devices. In the future we intend to add other

enlightened page sharing mechanisms—such as long-lived zero-page detection, page-table sharing and kernel text sharing—which will improve Satori's sharing discovery rate. We also intend to investigate the application of our technique to nearly-identical pages [4].

## Acknowledgments

We wish to thank members of the Systems Research Group at the University of Cambridge for the many fruitful discussions that inspired this work. We also wish to thank our shepherd, Geoffrey Voelker, and the anonymous reviewers for their insightful comments and suggestions that improved this paper.

## References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[2] 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*, 1997.

[3] P. J. Denning. The working set model for program behavior. In *Proceedings of the 1st ACM Symposium on Operating System Principles*, 1967.

[4] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *8th USENIX symposium on Operating System Design and Implementation*, 2008.

[5] S. M. Hand. Self-paging in the nemesis operating system. In *Proceedings of the 3rd USENIX symposium on Operating Systems Design and Implementation*, 1999.

[6] Hewlett-Packard Development Company, L.P. httperf homepage, 2008. `http://www.hpl.hp.com/research/linux/httperf/`, accessed 9th January, 2009.

[7] J. F. Kloster, J. Kristensen, and A. Mejlholm. On the Feasibility of Memory Sharing. Master's thesis, Aalborg University, June 2006.

[8] J. F. Kloster, J. Kristensen, and A. Mejlholm. Determining the use of Interdomain Shareable Pages using Kernel Introspection. Technical report, Aalborg University, January 2007.

[9] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Proceedings of the 3rd EuroSys conference on Computer Systems*, 2008.

[10] G. Neiger, A. Santoni, F. Leung, D. Rogers, and R. Uhlig. Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel® Technology Journal*, 10(3):167–178, Aug 2006.

[11] ObjectWeb Consortium. RUBiS – Home Page, 2008. `http://rubis.objectweb.org/`, accessed 9th January, 2009.

[12] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd USENIX symposium on Networked Systems Design and Implementation*, 2006.

[13] J. H. Schopp, K. Fraser, and M. J. Silbermann. Resizing Memory with Balloons and Hotplug. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.

[14] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J. Choi. Collaborative Memory Management in Hosted Linux Environments. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.

[15] L. H. Seawright and R. A. MacKinnon. VM/370 - A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[16] A. S. Tanenbaum. *Modern Operating Systems*, page 122. Prentice-Hall, 1992.

[17] (Unattributed). Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`, accessed 5th January, 2009.

[18] (Unattributed). Understanding Full Virtualization, Paravirtualization and Hardware Assist. Technical report, VMWare, Inc., 2007.

[19] (Unattributed). AMD-V™ Nested Paging. Technical report, Advanced Micro Devices, Inc., Jul 2008.

[20] (Unattributed). Performance and capacity requirements for Hyper-V, 2008. `http://technet.microsoft.com/en-us/library/dd277865.aspx`, accessed 9th January 2009.

[21] (Unattributed). *Resource Management Guide, ESX Server 3.5, ESX Server 3i version 3.5, VirtualCenter 2.5*, page 171. VMware, Inc., 2008. `http://www.vmware.com/pdf/vi3_35/esx_3/r35u2/vi3_35_25_u2_resource_mgmt%.pdf`, accessed 9th January, 2009.

[22] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating systems Principles*, 2005.

[23] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX symposium on Operating Systems Design and Implementation*, 2002.

[24] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[25] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th USENIX symposium on Operating Systems Design and Implementation*, 2002.

[26] W. Zhao and Z. Wang. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, 2009.