An Empirical Study of Memory Sharing in Virtual Machines

Sean Barker, Timothy Wood[†], Prashant Shenoy, Ramesh Sitaraman
University of Massachusetts Amherst
[†] *The George Washington University*
[sbarker,shenoy,ramesh]@cs.umass.edu, timwood@gwu.edu

Abstract

Content-based page sharing is a technique often used in virtualized environments to reduce server memory requirements. Many systems have been proposed to capture the benefits of page sharing. However, there have been few analyses of page sharing in general, both considering its real-world utility and typical sources of sharing potential. We provide insight into this issue through an exploration and analysis of memory traces captured from real user machines and controlled virtual machines. First, we observe that absolute sharing levels (excluding zero pages) generally remain under 15%, contrasting with prior work that has often reported savings of 30% or more. Second, we find that sharing within individual machines often accounts for nearly all (>90%) of the sharing potential within a set of machines, with inter-machine sharing contributing only a small amount. Moreover, even small differences between machines significantly reduce what little inter-machine sharing might otherwise be possible. Third, we find that OS features like address space layout randomization can further diminish sharing potential. These findings both temper expectations of real-world sharing gains and suggest that sharing efforts may be equally effective if employed within the operating system of a single machine, rather than exclusively targeting groups of virtual machines.

1 Introduction

Many modern computing services are hosted in large-scale data centers, where many separate applications are served simultaneously on thousands of machines. In the Infrastructure-as-a-Service model (IaaS), exemplified by services such as Amazon EC2, data center operators rent machine resources to customers, who then operate their own applications on their rented machines.

To facilitate servicing multiple customers on single servers, providers have turned to *virtualization*, in which multiple virtual machines (VMs) operate independently but are collocated on a single physical server. In such a model, memory efficiency is a key consideration, as it is often the bottleneck resource that determines the number of customers that providers can service on individual physical servers. Most virtualized systems simply partition available physical memory between VMs. This results in a hard cap of the number of customer VMs possible per machine based on the amount of memory consumed by each VM. If each customer can be serviced using only a small amount of memory, more customers can potentially be handled on a given server, resulting in greater utilization and lower hardware costs.

One technique for improving memory efficiency in virtualized systems is *content-based page sharing* (CBPS). In CBPS, duplicate blocks of memory (or 'pages') are collapsed into a single physical copy, with all duplicate virtual pages pointing back to the single physical page. This adjustment means that unneeded physical pages can be freed, lowering the memory footprint of the application or OS.

This form of memory deduplication has been experimented with in many virtualization platforms, and many of these systems have reported achieving impressive savings from sharing. For example, Difference Engine [6] reported absolute memory savings of 50% across three VMs (using page-level sharing), while VMware [19] reported savings as high as 40% across ten VMs.

These results are encouraging and appear to suggest that sharing can be used to great effect in real systems. However, there have been few studies of the real-world potential of page sharing across a variety of environments—i.e., in practice, how much sharing should we really expect to achieve? Other important issues have also received little scrutiny, such as the primary origin of sharing. Finally, there are some relevant questions about practices that may reduce sharing potential, such as randomizing or modifying memory contents for the sake of security. Practical investigation of these is-

sues is complicated by several factors, such as the need to tightly control experiments for comparison with other systems and the difficulty of gathering data from real-world machines. As a result, many systems have focused on comparing performance on benchmark workloads, which may not reflect real savings in practice.

In this paper, we conduct an empirical investigation of these questions about page sharing, under the assumption that all potential page sharing can be captured. This assumption is important because it separates the issue of evaluating a particular sharing system from evaluating the potential of page sharing itself. We conduct our investigation both on a set of memory traces captured from real-world users and on traces generated from a wide assortment of virtual machines.

Our major finding is that sharing tends to be significantly more modest (on the order of one half, or sometimes even less) than what might be expected from the literature. We also identify several interesting properties of page sharing, which both temper the expectations of sharing and may suggest alternative ways in which it should be deployed; these points are summarized below:

1. Self-sharing. We separate sharing into two essential categories: self-sharing (memory duplication within a single OS), and inter-VM sharing (memory duplication across multiple VMs). We find that in most cases, a significant majority (80% or more) of the sharing potential comes from self-sharing alone. This finding has important implications for the deployment of sharing systems. For instance, individual OS kernels may be nearly as well-suited to implementing sharing as conventional virtualized systems. Furthermore, such a shift would allow non-virtualized systems to take advantage of sharing.

2. Inter-VM sharing. Inter-VM sharing tends to be small, and effectively zero when the VMs are running heterogenous platforms. Sharing across VMs is more significant across a homogeneous platform, but even favorable conditions (e.g., cloned VMs) often derive a minority of savings (40% or less) from inter-VM sharing.

3. Sources of sharing. We provide a case study on the origins of sharing in a desktop environment by instrumenting the Linux kernel. Our system-aware memory tracing tool reveals how shared pages are used by programs – we find that the largest sharing source originates from GUI applications and related display libraries.

4. Security features and sharing. We find that OS security features, and address space layout randomization (ASLR) in particular, can reduce sharing by as much as 20%. Moreover, these features can reduce sharing across systems even when a high degree of homogeneity is imposed, such as in a virtual desktop environment.

In Section 2, we present background on sharing and motivation for why more careful study is needed. We detail our sharing model in Section 3 and describe our data and experimental results in Section 4. Our case studies on sharing sources and ASLR are presented in Sections 5 and 6, respectively. Finally, we mention related work in Section 7 and summarize our conclusions in Section 8.

2 Background and Motivation

Content-based page sharing has been targeted mostly at the hypervisor level within virtualized systems. In a virtualized system employing page sharing, the hypervisor (which has a global view of physical memory across all virtual machines) identifies sharable pages itself, then shares them without any input or cooperation from the virtual machine whose pages are being shared. In doing so, the primary aim is to capture sharing between VMs that are highly similar, significantly reducing the memory footprint of such VMs. This process is illustrated in Figure 1, in which six VM-level pages are serviced by only four physical pages.

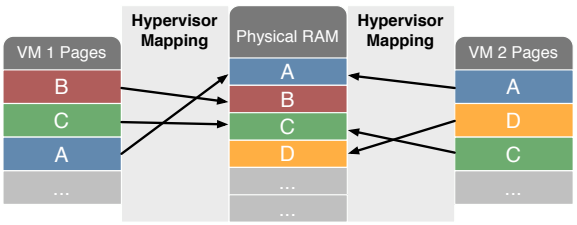

Figure 1: Page sharing between two VMs. The content of a page is indicated by its color/label.

Designing systems that can efficiently capture and exploit sharable pages has been the subject of a large volume of recent work. These efforts include both research systems ([6], [12], [17], [21]) as well as commercially available hypervisors ([2], [19]). Many of these systems report that 40% or more of a server's memory can be freed using page sharing techniques. These numbers are typically produced by grouping together several virtual machines on a host and reporting the total number of pages which could be freed due to sharing. However, this approach of simply tallying the total memory freed due to sharing does not offer any insights into how much memory is shared *internally to each VM* versus how much is shared *between VMs*.

During our investigations into improving page sharing in virtual environments, we uncovered a rarely discussed issue, namely that a large portion of the sharing found in virtualized environments is actually just internal to each VM. As a motivating example, we consider a simple benchmark used by other sharing systems ([6], [12]) – compiling the Linux kernel. Using two VMs booted from an identical Ubuntu 10.10 64-bit image, each with 1.5 GB memory, we compile the 2.6.32 kernel in each

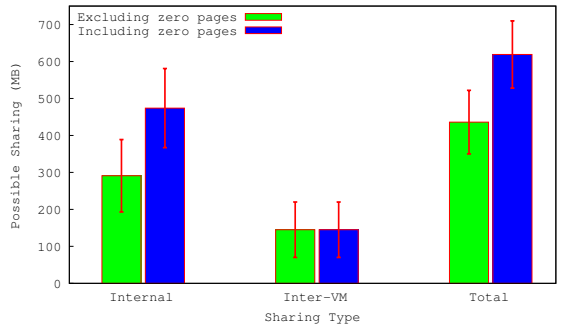

Figure 2: Sharing during kernel compilations in two identical Ubuntu VMs with and without zero pages.

(varying the minor version slightly to prevent caching of identical temporary files), and capture several memory snapshots spaced throughout each compile. We then consider each pair of snapshots across VMs and calculate the average internal sharing and average sharing across VMs. Figure 2 illustrates the total amount of sharing we found and the breakdown between internal "self-sharing" and inter-VM sharing. Excluding zero pages, we see that on average, 436 MB of the total memory could be shared, but of this, two-thirds (67%) was due to self-shared pages within individual VMs, with only the remaining third due to sharing between VMs. If we add in zero pages (which add only a single page to inter-VM sharing), the percentage of internal sharing rises to over three-quarters (77%).

This result runs counter to previous studies which have suggested that most page sharing is due to duplicated memory regions such as OS libraries that are common across multiple machines. Even in the previous example, in which the component VMs were identical and running nearly identical workloads, the majority of shared pages were sharable within isolated VMs. This has implications both for how memory sharing is managed, e.g., placing more VMs together may not significantly increase sharing levels, and how memory sharing is implemented, e.g., sharing within a single OS may be just as effective as attempting to do it across multiple VMs.

3 Types of Sharing

Let V be a virtual machine with a physical memory allotment m. For a given page size s, V has an array of m/s pages $P_V = \{p_1, p_2, \ldots, p_{m/s}\}$. Two pages p_i and p_j are *sharable* if their contents are byte-for-byte identical. For a given page p with k copies (including p), we can save $k-1$ pages by eliminating these duplicates and replacing them with references to the single copy p.

Let $UNIQUE(P_V)$ be the set of unique pages in P_V. Since this is the minimum number of pages needed to represent the memory of V, the **self-sharing** of V is the set of duplicate pages in P_V, given by

$$S_{self}(V) = P_V - UNIQUE(P_V)$$

Informally, self-sharing is simply the sharing that can be captured within a single virtual machine.

We can easily generalize this model to multiple VMs. Let G be the global set of VMs V_1 through V_k with corresponding page arrays P_{V_1} through P_{V_k}. The global page array is given by simply appending each component array: $P_G = \{P_{V_1}, P_{V_2}, \ldots, P_{V_k}\}$. We can then calculate $UNIQUE(P_G)$ and $S_{self}(G)$ as before. The **inter-VM sharing** of G is the total set of sharable pages in G *that cannot be captured by self-sharing alone*, given by

$$S_{inter}(G) = S_{self}(G) - \sum_{V \in G} S_{self}(V)$$

Informally, inter-VM sharing represents the sharing benefits that can be realized only by collocating the virtual machines in G. For a single machine V, $S_{inter}(V)$ is always zero, since all sharing is self-sharing. In general, if $S_{inter}(G) = 0$, then the VMs in G have no memory overlap—thus, no additional sharing can be captured by considering all VMs at once versus each VM in isolation.

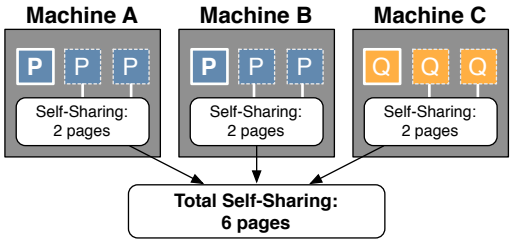

Figure 3: Sharing within single machines (self-sharing).

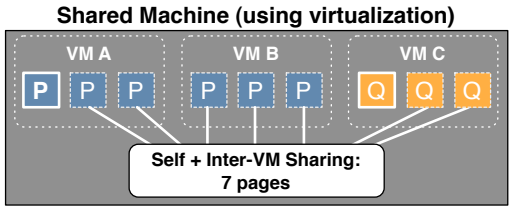

Figure 4: Inter-VM sharing between multiple machines.

We illustrate the difference between self-sharing and inter-VM sharing with a simple example, illustrated in Figures 3 and 4. Consider three machines A, B, and C that have sharable memory contents as shown in the figure: machines A and B both have three copies of page P and machine C has three copies of a different page Q. When we consider the machines individually, each can reduce its three pages down to one using self-sharing, for a total of six pages saved (two pages per machine). However, if A, B, and C are virtual machines residing on the

same physical host, as shown in Figure 4, the hypervisor may share a single copy of P globally for both VMs A and B—freeing one additional page compared to the self-sharing case. However, the shared page Q sees no benefit because the page is distinct only to VM C. In total, grouping the virtual machines together only results in an increase of one shared page which could not be found using self-sharing alone.

3.1 Sharing Properties

The model above illustrates how sharing can occur both within a single physical (or virtual) machine and between multiple virtual machines. Thus, we stress that while inter-VM sharing is only applicable to virtualized systems, self-sharing is just as possible in conventional systems. This distinction is important because most related work on sharing has focused heavily on virtual machines – this implicitly asserts that inter-VM sharing is the primary component of sharing.

Based on our sharing model, we can reason about when each type of sharing is likely to be more important. If individual machines have mostly unique pages, but there are many such machines (with some overlap), then hypervisor-level sharing – inter-VM sharing – is clearly favored. For example, if the memory of VM A is comprised of 10 distinct pages, and the memory of VM B is comprised of the same set of 10 pages, then inter-VM sharing can save 10 total pages (a 50% savings), while individually A and B could not achieve any savings via self-sharing. However, as the amount of self-sharing (duplicates within a single machine) rises, the relative importance of inter-VM sharing is likely to fall significantly. For example, if we take A and B and simply triple their memory contents (3 copies each of 10 distinct pages), then inter-VM sharing still provides the same 10 pages, but self-sharing provides 20 pages each, for a total of 40 pages – a 67% savings on top of which inter-VM sharing provides a modest 17% savings. In general, the potential of inter-VM sharing is bounded by the number of machines involved. This is because, for a given page p, inter-VM sharing can only save (at most) one page per machine. All other sharing involving p will be captured by self-sharing, and this amount of sharing is only bounded by the total number of pages available: k additional copies of p can add up to k shared pages (if they are all within the same machine). Inter-VM sharing is only likely to be more significant than self-sharing when there are many VMs with minimal internal redundancy.

3.2 Calculating Sharing

Our model is idealized in that real systems usually cannot actually capture *all* possible sharing. Some sharing opportunities may be short lived due to changing memory contents, and may either be missed by the system or passed over intentionally to reduce overhead. New systems are constantly attempting to identify both the largest and safest (long-lived) possible sharing opportunities, but we would like to sidestep this issue and simply tally all possibilities. Since proposing a deployable system is outside the scope of this work, we use a simple, heavyweight, but effective method of calculating sharing: scanning and snapshotting. Scanning refers to simply sequentially reading memory contents, hashing each page of memory, and outputting the list of hashes to be checked for duplicates. Since duplicate pages have duplicate hashes, checking for equality is trivial. Snapshotting refers to pausing the VM while scanning, which prevents memory from changing while looking for duplicates.

Scanning is used by some commercial systems such as VMware[19]—however, in order to prevent excessive overhead, scans can only be performed at moderately-spaced intervals (generally on the order of minutes), and may miss sharing due to changing memory. Since we are not concerned with efficiency, we simply suspend the VM, scan the memory snapshot, then resume. From the VM's perspective, this reduces the scan time to zero, so we can (in theory) perform scans as rapidly as desired. A second benefit of this approach is that since we read from the raw binary contents of memory, we can divide it into pages of any desired size.

One notable extension of the standard content-based sharing approach (proposed in [6]) is the use of small deltas between memory. This allows sharing pages even when they differ by a small amount, as opposed to restricting to when pages are exactly identical. We note that this type of sharing can be approximated by considering pages of smaller and smaller sizes, although decreasing the page size is only practical to a point, after which the overhead of handling an increasing number of pages becomes excessive.

Finally, our measurements of sharing explicitly ignore pages filled completely with zeroes. Zero pages are plentiful in many scenarios because the operating system and applications may zero out pages for future use; however, for the same reason, they are usually regarded as poor candidates for sharing. The intuition is straightforward – zero pages are plentiful but not likely to remain zero pages for long. Satori (being more concerned with sharing duration than most other projects) reported confirmation of this [12], and reported 20 times as much sharing from zero pages as from nonzero pages, but also reported that this page sharing was quite short lived. In our investigations, we adhere to the notion that zero pages are not a highly useful form of sharing, and disregard them in our experiments. This is important to note because including zero pages tends to greatly exaggerate results.

4 Evaluation

In this section, we present a study of sharing in actual machines and VMs. We pay particular attention to self-sharing versus inter-VM sharing in these systems, as well as exploring the effect of changing the underlying system (e.g., operating system version or application setup).

4.1 Data Collection

Our memory traces come from two sources – one set of traces captured from real-world machines, and a second set of traces generated from synthetic experiments. This both gives a holistic picture of real-world sharing potential and allows us to explore a wide set of systems.

Real-world memory traces. Our real-world memory traces come courtesy of the Memory Buddies project [21], which deployed a memory tracer onto a variety of UMass departmental machines and made these traces publicly available [11]. We focus on a set of traces gathered over a weeklong period from seven machines. The seven machines included four MacBooks (used by individual users) running similar versions of Mac OS X and three Linux servers running a variety of server tasks (web, mail, ssh, etc). The MacBooks each had 2 GB of RAM, and the Linux servers had 1, 4, and 8 GB of RAM. These are all production machines handling actual user workloads. Each machine produced a memory trace every thirty minutes during the time it was powered on (150 to 350 traces per machine), resulting in roughly 1700 traces in all. Each trace is comprised of one hash value per physical page (4 KB of memory).

Generated memory traces. To supplement our real-world traces, we configured a set of custom virtual machines. We produce a memory trace from a VM by suspending the VM, then reading the resulting binary memory image to create the hash list. This has the advantage (versus running a VM-based tracing tool) of having no impact on memory within the VM during tracing. Furthermore, since we can access the full binary contents of memory, we can use any sharing granularity desired. Sequential traces are produced by a series of suspend-resume cycles, and resetting the VM to a previous snapshot allows resetting memory to an exact previous state to evaluate the impact of a subsequent action or workload. We configured 10 distinct VMs, each with 1.5 GB physical memory:

- **Linux**: Ubuntu 10.04 and 10.10, and CentOS 5.3 (no GUI for CentOS). These distributions were chosen to be representative of typical desktop `apt` and server `rpm` based distributions. We consider both 32-bit and 64-bit versions (6 VMs total).
- **Windows**: Windows XP (x86) and Windows 7 x86 and x64 (3 VMs total).
- **Mac**: Mac OS X 10.6 Server (Snow Leopard). The server version is used due to virtualization restrictions, but is very similar to the desktop version.

For each of the 10 VMs, we use three application setups for capturing memory traces:

- **No applications**: The VM is freshly booted, but is not running any further software.
- **Server applications**: The VM runs a typical LAMP stack: Apache 2, PHP 5, and MySQL 5. The workload consists of issuing a series of MySQL queries and serving a mix of static and dynamic pages.
- **Desktop applications**: The VM runs a typical set of desktop applications: web browser (Firefox), office applications (OpenOffice), email client, media player, etc. The workload consists of opening several web pages and office documents (text document, spreadsheet, etc.) and playing a media file.

For each pair of VM and application setup, the VM is booted, the applications are loaded and the workload executed, then a memory snapshot is captured. This resulted in 28 traces total (no desktop traces for CentOS due to the lack of an installed GUI).

4.2 Sharing in Real-World Traces

We examined the real-world traces to determine if self-sharing comprised a substantial portion of total sharing. Since sharing can change substantially over time, we processed each set of sequential traces (for each of the seven host machines) and calculated the min, max, and average potential sharing over the weeklong recording period.

As discussed previously in Section 3.2, we exclude zero pages from all presented results, including both sharing percentages and absolute sharing. While we believe this decision more accurately reflects useful sharing, including zero pages in our results would have only a modest impact – for example, in nearly all cases, sharing from zero pages in our real-world traces comprises less than 5% of the total memory.

4.2.1 Self-Sharing

When we consider each machine individually, the sharing observed is entirely self-sharing. The amount of self-sharing in each machine is shown in Figure 5, both as a percentage of machine memory and as absolute sharing in MB. We see a modest, but not insignificant, level of self-sharing present in most of the machines – on average, about 14% of the total memory in any machine was sharable at any time. This demonstrates that a significant amount of duplicate memory is present even in isolated (non-virtualized) machines. Another interesting feature

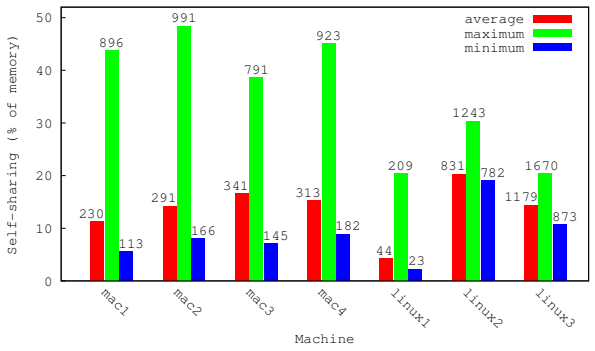

Figure 5: Self-sharing in the real-world memory traces, with absolute shares (in MB) noted.

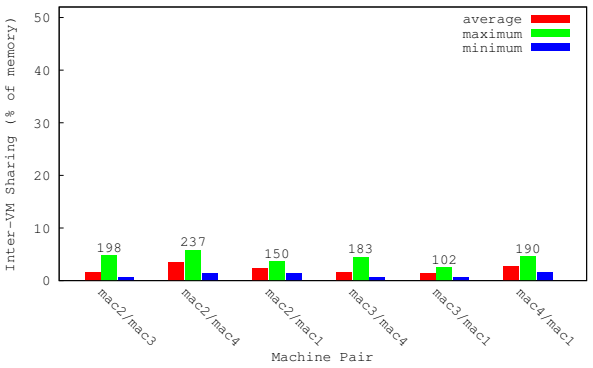

Figure 6: Inter-VM sharing between real-world machine pairs, with maximum absolute shares (in MB) noted.

is a difference of up to an order of magnitude between the minimum and maximum sharing observed, demonstrating the volatility of sharing over time. Since the average is much closer to the minimum than the maximum, this suggests brief periods of very high sharing potential surrounded by long periods of much lower sharing potential. Finally, we observe that the sharing variation is greater in the Macintosh machines than in the Linux machines – we speculate that this may be due to the fact that the desktop machines see a much wider range of applications compared to the servers which have a fixed purpose.

4.2.2 Inter-VM Sharing

Next, we consider the inter-VM sharing between pairs of machines represented in the real-world traces. For a given pair of machines (M_1, M_2), we wish to calculate both the self-sharing and inter-VM sharing between these machines. However, since there are hundreds of traces from each machine to select for (M_1, M_2), evaluating every possible trace pairing is infeasible. Thus, we simply randomly select several hundred pairs of traces from M_1 and M_2, then calculate the min, max, and average inter-VM sharing for (M_1, M_2) using this set. We perform this procedure for every possible (M_1, M_2) pair – since there are 7 machines, there are 21 total machine pairings.

The results for all Mac/Mac machine pairings are shown in Figure 6 – *for all other pairings*, including both Mac/Linux and Linux/Linux pairings, the inter-VM sharing observed was negligible (always under 1% of the total memory and usually under 0.1%). Even in the case of Mac/Mac pairings, as seen in Figure 6, the average inter-VM sharing is strikingly low, comprising only 2 to 3 percent of the total memory. Furthermore, even considering two traces selected for the greatest possible inter-VM sharing, this sharing never exceeded 6% of the total.

In the cases of greatest average inter-VM sharing, the machines involved showed average self-sharing in the 10 to 15 percent range. This means that even in the observed cases most favorable to inter-VM sharing, this sharing still only accounted for a mere 20% of the total sharing potential, with the other 80% attributable to self sharing. In the case of other pairings (which displayed effectively no inter-VM sharing), the memory gained from collocating only these machines would be negligible.

Considering more than 2 machines together naturally increases inter-sharing. However, we find the trends to be similar even when considering larger machine tuples. For example, if we consider traces from all 7 machines at once, we find a typical total sharing around 15% – however, *over 90%* of this sharing is solely attributable to self-sharing, with less than 10% being added by inter-VM sharing. In other words, even collocating all seven of these machines for the purpose of memory sharing would save very little – almost all possible sharing could be captured simply by having each machine eliminate duplicates within its own memory.

These results suggest that page sharing systems should not exclusively be directed at hypervisor-level systems such as Xen and VMware, but could also have worthwhile benefits when implemented in a commodity operating system such as Windows or Linux. Doing so would enable regular home users to reap the benefits of page sharing, and would allow virtual machines to maintain closer control over their memory usage. Moreover, we find that the added benefit from focusing sharing on virtualized systems may be substantially less than expected.

4.3 Platform Homogeneity

A set of VMs with a homogeneous underlying platform will have more sharing potential than a set of heterogenous machines, but the degree of homogeneity can vary widely. For example, a Windows 7 system may be quite different from an Ubuntu Linux system, but fairly similar to a Windows XP system. We can also consider factors such as architecture – for example, are Windows 7 x86 and Windows XP x86 closer or further apart than Windows 7 x86 and Windows 7 x64? As another exam-

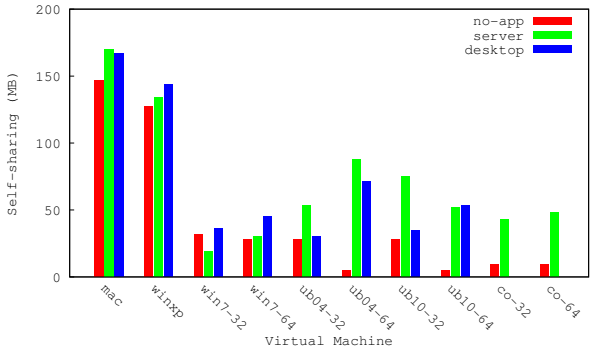

Figure 7: Self-sharing in the virtual machine traces.

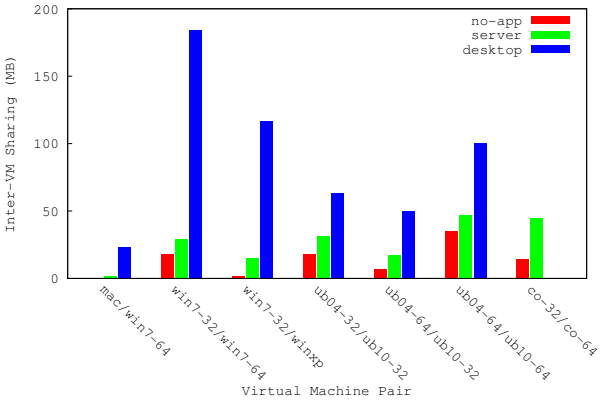

Figure 8: Inter-VM sharing between VM pairs.

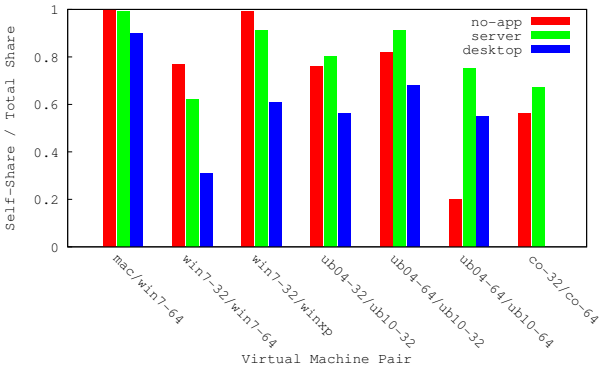

Figure 9: Self-sharing over total sharing among VM pairs.

ple, are the operating systems or user applications more important to compare with regards to sharing? We investigate these questions using our virtual machines and memory traces generated as detailed in Section 4.1.

4.3.1 Self-Sharing

As with our real-world traces, we first investigate the self-sharing present in our VMs. For each of the three application setups, the amount of self-sharing present in each VM is shown in Figure 7. The most notable feature seen is that the sharing in Mac OS X and Windows XP is much greater than in Windows 7 or any of the Linux VMs. This suggests that these systems tend towards a higher degree of internal memory redundancy, and may see greater benefit from harnessing this redundancy. Additionally, we see that in these cases, sharing does not substantially increase from the 'no-app' case when adding applications – this indicates that the base system is providing the bulk of the self-sharing, with only slight increases from the user-level applications.

Another interesting feature seen is that the base system sharing in both versions of Ubuntu decreases significantly when switching from a 32-bit system to a 64-bit system. Presumably, this indicates a lower level of redundancy in the 64-bit system libraries than in their 32-bit counterparts. Sharing in both versions of CentOS is quite low – we believe (based on observations discussed in Section 5) that this is largely due to the lack of a GUI.

4.3.2 Inter-VM Sharing

We next consider sharing between pairs of VMs. For every possible pairing, we calculate both the self-sharing and inter-VM sharing for each of the three application setups. While the complete results for all pairings are omitted for brevity, a representative sample is shown in Figures 8 and 9. Figure 8 shows the absolute inter-VM sharing for each of the selected pairings, while Figure 9 shows the relative importance of inter-VM sharing compared to self-sharing in each pairing.

Major OS mixes. Our first major observation is that inter-VM sharing is negligible in all mixes of major OSes (Mac OS X, Windows, or Linux). In all cases, these pairings closely resemble the `mac/win7-64` pair as seen in Figure 8 – there is essentially no inter-VM sharing in the base systems, and only a very small amount in the application setups. What little application sharing is present is likely due to shared resource files, hence why the sharing is significantly larger in the desktop case than the server case. Even with a common application setup, however, the inter-VM sharing represents only a minor fraction of the total sharing – less than 10%, with the rest attributable to self-sharing alone, as seen in Figure 9.

OS version mixes. Our second observation is that with different versions of a common major platform (e.g., Ubuntu 10.04 vs 10.10), sharing is significantly reduced in the base system, but is not strongly reduced in the case of common applications. The best example of this is seen in the `win7-32/winxp` pairing – although there is almost no inter-VM sharing in the base system ($< 5\%$), the amount of inter-VM sharing increases significantly in the application setups (roughly 40% inter-VM sharing in the desktop case). This is due to the fact that many of the applications themselves (e.g., OpenOffice)

run the same version on these systems, adding a significant amount of sharing. This behavior is also seen in the various Ubuntu pairings – mixing versions significantly degrades inter-VM sharing in the base system (e.g., 75% self-sharing between Ubuntu 10.04 and 10.10 32-bit), but these cases still show significant application sharing.

Architecture mixes. Overall, we see similar behavior when changing OS architecture (32-bit to 64-bit or vice versa) as when changing the OS version. However, changing the architecture is still significantly less disruptive than changing the major OS version, which completely eliminates most sharing.

The one notable case in which we see an architecture-specific behavior is when pairing the two 64-bit Ubuntu versions (`ub04-64/ub10-64`). In this case, almost all sharing in the base systems is due to inter-VM sharing (80%) rather than self-sharing – this is likely due to the fact that both of these systems displayed minimal self-sharing themselves, as seen previously in Figure 7.

Application types. In all cases (except CentOS, which did not run a GUI), we see that sharing was substantially higher in the GUI desktop applications than in the server applications. This may be partially due simply to the higher memory footprint of our desktop applications, but is also likely due to the tendency of GUI-related libraries to increase memory redundancy. We explore this tendency in Section 5.

4.4 Variable-Sized Hashing

Sharing is typically done on a page-by-page basis (that is, only sharing at the granularity of an entire page). However, one can also share on a different granularity, trading off between sharing potential and efficiency – a smaller granularity increases overhead, but is capable of sharing smaller chunks of memory. Since operating systems allocate memory on a per-page basis, it is most natural to consider even multiples (0.5, 2, 4) of the base 4 KB page size. Again, however, there is no requirement to share using these granularities. Thus, we examined several of our traces with sharing granularities varying from 0.4 to 2.4 in intervals of 0.1. The results for a typical trace (taken from an Ubuntu VM) are shown in Figure 10.

As expected, sharing increases modestly as the granularity increases. We also note the significant peaks at 0.5, 1, and 2 hashes per page (the evenly-dividing settings). The greatest ratio of sharable memory to hashes per page (a proxy for processing overhead) is still at the standard 1-page granularity. This is in line with other reported results [21] that have suggested modest but diminishing returns from increasing the sharing granularity. Furthermore, these results confirm that the only reasonable granularities evenly divide the page size, as other granularities significantly reduce possible sharing.

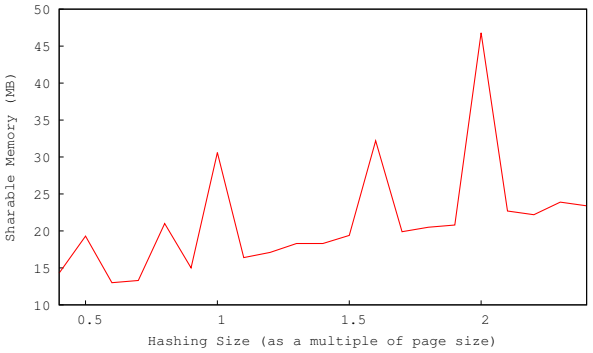

Figure 10: Self-sharing with variable hashing sizes.

5 Sources of Self-Sharing

Our previous results demonstrate the importance of self-sharing, but do not explain where this self-sharing originates. The root of self-sharing – internal redundancy – presumably only exists by accident. However, we have seen that that redundancy is common in all systems studied. To shed light on what causes redundancy, we have conducted a case study on Linux desktop applications.

5.1 An Extended Memory Tracer

Since identifying the source of sharing requires more information than basic memory traces, we extended the memory tracer (a kernel module) used to collect our real-world Linux traces. The original tracer simply walks through each page of memory and calculates a hash based on the page contents. For pages in use by active processes, our extended tracer also collects two additional pieces of information:

- The content type of the page – either a specific part of a regular program address space (e.g., stack, heap), or a mapped page of a shared library.
- The process(es) using the page. For a shared library page, there may be any number of processes using the page. For other pages, there will only be one process using the page.

For example, two pages might give the following (omitting the memory content hash values):

```
[libc-2.12.so 000b6000 r-xp]: sshd apache2
[heap]: mysqld
```

The first is a specific page of `libc`, in use by SSH and Apache. The second is a page in the MySQL heap. Using this extended information, we can calculate not only the amount of sharing possible, but which processes or libraries are actually involved in the sharing.

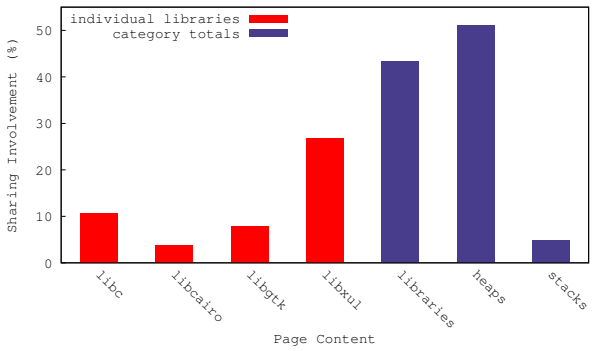

Figure 11: Sharing by content as a percentage of total sharing.

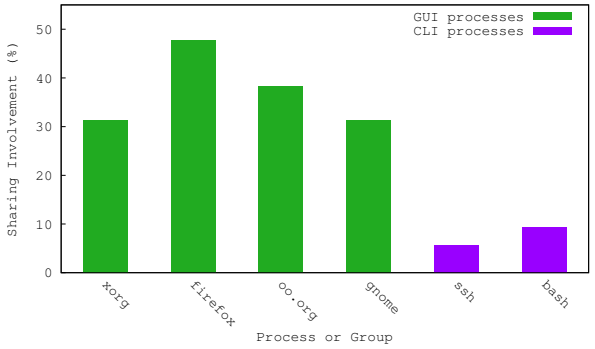

Figure 12: Sharing by process as a percentage of total sharing.

5.2 Case Study: Desktop Applications

A growing trend in virtualized desktop systems is the use of thin clients that communicate with homogeneous server-side VMs. To explore this potentially lucrative source of sharing, we conduct a case study of desktop applications in an Ubuntu VM. We examine sharing opportunities by page type, then discuss some of the main applications which exhibit high levels of sharing.

The primary page types we examine are stack pages, heap pages, and shared library pages. The results for each of these, as well as for a few individual shared libraries, are shown in Figure 11 (excluding zero pages as before). Note that these results are not additive, since multiple page types (e.g., a stack page and a heap page) may have the same content, which causes both to be included in the same chunk of sharing.

We see that the largest single source of sharing is heap pages, which are involved in over 50% of all sharing within the VM. Library pages are close behind, involved in 43% of all sharing. Stack pages, in contrast, are relatively minor, at less than 5% involvement. Of the shared libraries, the single library involved in the most sharing is `libxul`, the user interface library used by Firefox. Other libraries, such as `libc`, are used more widely, but were involved in substantially less sharing overall.

We also looked at sharing involvement by process rather than content. Selected processes are shown in Figure 12 (note that these results are again not additive, since multiple processes may be using a single page). The single most important process to sharing was Firefox, which was involved in nearly half of all sharing – this is understandable given the importance of `libxul` to sharing seen in Figure 11. However, we also see substantial sharing in other GUI applications such as OpenOffice (38%) as well as system-wide GUI processes such as the X server (31%) and all Gnome-related processes (also 31%). In contrast, headless applications such as SSH (6%) were involved in much less sharing.

The importance of GUI applications to sharing is further reinforced when we examine the most widely shared individual pages (i.e., those pages with the most copies). Every one of the top six shared pages, which alone are responsible for almost 10 MB of sharing, are used by Firefox, Xorg, or both. The single most shared page, with 597 distinct copies (2.3 MB shared) was a heap page used by Xorg. These situations are likely byproducts of repeatedly allocating and freeing a particular data structure, since copies may include old pages that have yet to be reclaimed by the OS as well as pages currently in use.

6 Memory Security and Sharing

Operating system designers are constantly seeking new ways to harden their systems against attackers and reduce exploit opportunities. One common attack vector involves overwriting memory contents (e.g., a buffer overflow attack), particularly when the overwritten memory is at a known address, such as a key library function. To harden systems against these kind of attacks, modern operating systems employ a technique known as **address space layout randomization**, or ASLR, in which the location of key libraries and program assets (e.g., the heap) in memory are randomized. This randomization is designed to prevent attackers from making use of known addresses (e.g., in corrupting targeted data). For the example address space shown in Figure 13a, two possible randomizations are shown in Figures 13b and 13c.

While ASLR is exclusively a security measure, since it modifies the contents of memory, it has the potential to affect the level of page sharing possible. Two VMs that may be running identical software may have different memory contents if ASLR is enabled, resulting in less possible sharing. For example, the randomized address spaces shown in Figures 13b and 13c may share less than if they were not randomized (and hence equal to Figure 13a). We conduct a study on the impact of ASLR on sharing potential by evaluating multiple implementations across several operating systems.

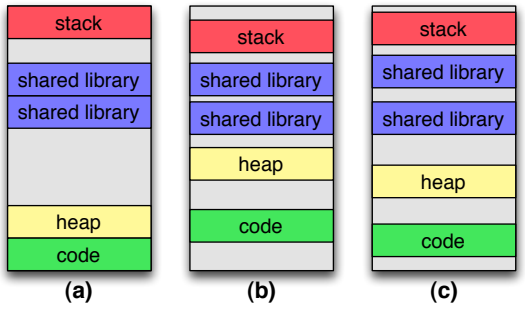

Figure 13: A non-randomized address space (a) and two examples of the address space after randomization (b and c).

6.1 Current ASLR Support

While ASLR adoption has been gradual and the level of support varies among operating systems, most popular OSes have at least rudimentary support and are moving towards more complete instrumentation. Most implementations allow for enabling or disabling randomization, which we exercise in studying its impact. We selected four implementations for study – two in Linux, one in Windows 7, and one in Mac OS X Lion.

Linux. The Linux kernel first introduced ASLR support in 2005, and modern versions randomize the major components of a process (library locations, stack, heap, code) [8]. The ability to toggle system-wide ASLR is provided via the `/proc` interface, as well as an intermediate setting in which heap randomization is not used, but all other randomizations are.

PaX. An alternate implementation of ASLR for Linux is provided by PaX [13], which is a patch for the mainline Linux kernel aimed at improving overall security. A PaX-enabled kernel is used by several 'hardened' distributions of Linux aimed at maximizing security, and can also be deployed in most normal distributions. The ASLR implementation in PaX provides several features not provided by the standard Linux implementation, such as randomization within the kernel itself [7].

Windows. Microsoft introduced ASLR support in Windows Vista and continued in Windows 7, providing randomization of the stack, heap, DLLs, and so forth [20]. ASLR is enabled on a per-application basis, and is opt-in by default. While most system-provided applications within Windows enable ASLR, third-party application support has been slow [16]. However, Microsoft recently released a utility [5] that provides the ability to forcibly enable or disable ASLR for particular processes. In our tests, we encountered no ill effects from enabling it for applications that do not opt-in by default.

Mac OS X. Apple first introduced a simplistic form of ASLR in Mac OS X 10.6 (Snow Leopard), and support was expanded in 10.7 (Lion) [9]. Unfortunately, there is presently no straightforward way to disable randomization within Lion – the only known method [10] relies on setting a particular POSIX flag during process creation. To leverage this, we write a script that simply sets this flag, then spawns the target application (which runs without randomization).

6.2 Evaluating ASLR's Sharing Impact

For each of the four ASLR implementations, we wish both to identify whether randomization has an impact on sharing, and if so, to determine the extent of this impact. To do this, we simulate a scenario in which many VMs are booted from an identical base image. This is a lucrative scenario both for virtualization and for page sharing, and represents a case in which users are likely to care about fine-tuning sharing potential. We run this scenario for each of the three host operating systems – Ubuntu 10.10 64-bit (for both the standard Ubuntu kernel and a patched PaX kernel, both version 2.6.32), Windows 7 64-bit, and Mac OS X 10.7.

For a single OS, our test procedure (using a single VM) is as follows. First, we globally disable ASLR, using one of the tools mentioned in the previous section (note that in the case of Mac OS X, we cannot disable system randomization). We then reboot the VM to reset memory to a reliable state. Then, we populate the VM's memory by opening a predefined list of applications (web browser, text editor, office software, music player, etc.) using a shell script or batch file. After letting the contents of memory settle, we capture this 'non-randomized' memory snapshot. We then globally enable ASLR, reboot, and then repeat the snapshot procedure again to obtain a 'randomized' snapshot.

To simulate booting multiple VMs from the same image, we repeat this four times, resulting in a set of four randomized snapshots and a set of four nonrandomized snapshots. Since the only substantive difference between the sets is whether randomization is used, any significant reduction in sharing in the randomized snapshot set should be due to ASLR – furthermore, the use of multiple snapshots averages any other memory differences that occur between reboots.

The results, as a percentage reduction in sharing, are shown in Figure 14. The total sharing reduction is further broken down into self-sharing and inter-VM sharing – note that these are not additive, since they do not contribute equally to the total sharing. We see a modest, but noticeable reduction in sharing across all implementations. The largest reduction is seen in Windows 7, in which total sharing was reduced by 13% (in line with [18], which reported a 16% reduction in Windows 7). Total sharing in Mac OS X was reduced by only 3% – however, as noted above, randomization was not disabled for the system itself, and hence this result is conservative.

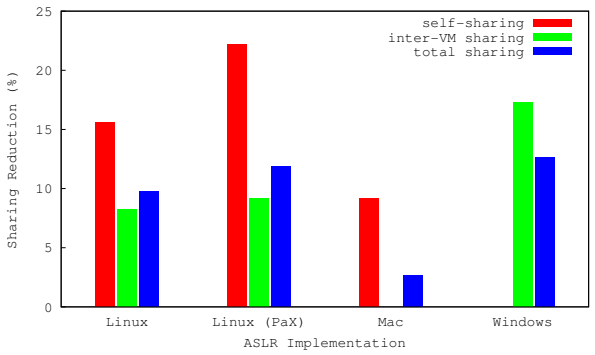

Figure 14: Reduction in page sharing observed with a variety of ASLR implementations.

Overall sharing in Linux was reduced by 10% using the stock kernel, while the PaX-patched kernel reduced sharing by a slightly higher 12%.

The results of the sharing breakdown are somewhat more surprising. In the case of Linux, ASLR reduced sharing both within single systems and across systems. However, in the case of Windows, the entire reduction was due to inter-VM sharing – individual systems shared the same amount regardless of randomization. This suggests that in a non-virtualized environment, sharing in Windows is not impacted at all by ASLR. Oddly, we observe the opposite behavior in Mac OS X – here, all reductions were due to self-sharing, with no reduction observed in sharing across systems. One reason for this may be the relative amount of self-sharing – as we observed in Section 4.3.1, self-sharing in Mac OS X was much higher than in Windows 7.

7 Related Work

One of the first systems implementing page sharing was the Disco virtual machine monitor (VMM) [3], which proposed sharing identical code pages (typically read-only) between virtual machines, as well using as a shared copy-on-write disk to share disk contents between VMs. For sharing memory, the original technique of content-based page sharing was introduced in VMware ESX Server [19]. CBPS has received significant attention because it requires no assistance from the guest operating system, and thus can be performed transparently within the virtualization layer.

The original virtual-machine centric nature of most page sharing work has persisted past the original VMware and Disco papers, and provides an important motivational basis of most page sharing systems today. One possible reason for this (besides simple precedent set by earlier papers) is that it seems intuitive that groups of virtualized systems could provide significant gains that cannot be realized within a single operating system. In contrast to this focus, we argue that page sharing need not be targeted exclusively at virtualized environments.

Several systems have built on the core CBPS idea to further exploit potential savings. The Potemkin VMM [17], which was originally built on Xen, makes heavy use of page sharing by creating new virtual machines as clones of an existing VM image. The clones are then able to store only differences from the base image rather than allocating their own dedicated memory space, in effect replacing a memory partition with a single base image and a set of memory deltas. This is an example of a special type of virtualized environment which is undoubtedly well suited to page sharing, since a new VM is (by definition) initially able to share 100% of its pages with the base VM. We explore several setups of this type and find that it is the exception to our general observation that self-sharing from individual virtual machines comprises the vast majority of sharing in groups of machines.

Another extension of the base CBPS technique has been sub-page sharing, in which full memory pages are broken down into pieces to allow for finer grained sharing. This technique was applied both in Difference Engine [6] (based on Xen) and Memory Buddies [21] (based on ESX Server). Difference Engine also went a step further and considered storing small 'patches' between memory, rather than just uniform sub-page sharing. Additionally, Difference Engine employed compression of non-shared VM memory to increase overall memory efficiency. However, they reported that the majority of sharing benefits were attributable to the basic page sharing paradigm rather than additional enhancements. Intelligent collocation of multiple VMs to optimizing sharing was explored using memory fingerprints in [21] and hierarchical tree models in [15].

A system for sharing memory in massively parallel applications using a distributed hash table was proposed in [22]. This work also considers the distinction between inter-node and intra-node sharing in parallel applications (in which inter-VM sharing is relatively more significant). We complement this work by focusing on sharing in general purpose, single-node virtualized systems.

The Satori [12] system took a different approach to sharing altogether and argued that most potential sharing lasts only a few seconds rather than at least a few minutes. A typical CBPS approach involving periodically scanning memory is insufficient for capturing this type of sharing, since scans cannot be performed on the granularity of a few seconds for performance reasons. Satori implemented sharing by watching for identical regions of memory when read from disk – this is also the approach that was integrated into Xen. However, the downside to this approach is that it requires modifications to the guest operating systems themselves. This represents a trade off between transparent sharing requiring guest modifi-

cations (the Satori/Xen approach) and oblivious sharing requiring no such modifications (the VMware approach).

Recent versions of the Linux kernel include sharing functionality via the KSM kernel module [1], which uses a scanning approach similar to that of VMware. While originally developed for sharing within the Linux KVM virtualization infrastructure, KSM can also be used for sharing within a nonvirtualized Linux system. An evaluation of KSM in a virtualized setting is given in [4]. The Singleton system [14], also based on KSM, focused on optimizing caching for low overhead deduplication.

8 Conclusion

Page sharing presents an opportunity to increase memory efficiency, but understanding the dynamics affecting page sharing are important to maximizing its benefits. We present an investigation and analysis into the sources of page sharing, and find that in many cases, the majority of sharing potential is attributable to redundancy within single machines (self-sharing) rather than between multiple machines (inter-VM sharing). This suggests (1) that sharing may be effectively exploited at the level of a single VM rather than a hypervisor, and (2) that sharing need not be restricted to virtualized systems at all.

For sharing across VMs, we investigate several application platforms and find that operating system homogeneity is the most important component of inter-VM sharing, with application, architecture, and version homogeneity of lesser (but still significant) importance. In particular, we see effectively no sharing between different base platforms – e.g., between a Windows and Linux system. Inter-VM sharing is still present, but greatly reduced, by changing the version of the base system.

We also conduct a case study of self-sharing within the Linux kernel and find that GUI applications and associated system libraries are the clearest source of self-sharing potential. We leave porting our tracing tool to non-Linux systems as future work to investigate whether similar trends will hold in other operating systems. Finally, we explore the impact of address space layout randomization on sharing potential. We find that in all major systems, this feature has a measurable negative impact on sharing potential. We are continuing to explore the factors behind both favorable and unfavorable sharing scenarios, and believe that understanding these issues will enable more efficient memory usage in both virtualized and nonvirtualized systems.

Acknowledgements. We would like thank our shepherd, Haibo Chen, and the anonymous reviewers for their valuable comments and feedback. This work is supported by NSF grants CNS-0519894, CNS-1117221, CNS-1143655, CNS-0916972, and CNS-0855128.

References

[1] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using ksm. In *OLS* (July 2009).

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP* (Oct. 2003).

[3] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. In *TOCS* (Nov. 1997).

[4] CHANG, C.-R., WU, J.-J., AND LIU, P. An empirical study on memory sharing of virtual machines for server consolidation. In *ISPA* (May 2011).

[5] The enhanced mitigation experience toolkit. `http://support.microsoft.com/kb/2458544`, 2011.

[6] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: harnessing memory redundancy in virtual machines. In *OSDI* (Dec. 2008).

[7] Kernel hardening roadmap. `https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening`, 2012.

[8] Linux kernel aslr implementation. `http://xorl.wordpress.com/2011/01/16/linux-kernel-aslr-implementation/`, 2011.

[9] Mac os x lion security. `http://www.apple.com/macosx/whats-new/features.html#security`, 2011.

[10] How gdb disables aslr in mac os x lion. `http://reverse.put.as/2011/08/11/how-gdb-disables-aslr-in-mac-os-x-lion/`.

[11] Memory buddy trace repository. `http://traces.cs.umass.edu/index.php/CpuMem/CpuMem`, 2009.

[12] MIŁÓS, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: enlightened page sharing. In *USENIX ATC* (June 2009).

[13] Pax. `http://pax.grsecurity.net/`, 2012.

[14] SHARMA, P., AND KULKARNI, P. Singleton: System-wide page deduplication in virtual environments. In *HPDC* (June 2012).

[15] SINDELAR, M., SITARAMAN, R. K., AND SHENOY, P. Sharing-aware algorithms for virtual machine colocation. In *SPAA* (June 2011).

[16] Dep/aslr implementation progress in popular third-party windows applications. White paper, Secunia Research, 2010.

[17] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP* (Oct. 2005).

[18] Project vrc (virtual reality check). `http://www.projectvrc.com/white-papers`, 2010.

[19] WALDSPURGER, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36* (Dec. 2002).

[20] WHITHOUSE, O. An analysis of address space layout randomization on windows vista. White paper, Symantec Advanced Threat Research, `http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf`, 2007.

[21] WOOD, T., TARASUK-LEVIN, G., SHENOY, P., DESNOYERS, P., CECCHET, E., AND CORNER, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *VEE* (Mar. 2009).

[22] XIA, L., AND DINDA, P. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *VTDC* (June 2012).