DOI:10.1145/1831407.1831429

Difference Engine: Harnessing Memory Redundancy in Virtual Machines

By Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat

Abstract

Virtual machine monitors (VMMs) are a popular platform for Internet hosting centers and cloud-based compute services. By multiplexing hardware resources among virtual machines (VMs) running commodity operating systems, VMMs decrease both the capital outlay and management overhead of hosting centers. Appropriate placement and migration policies can take advantage of statistical multiplexing to effectively utilize available processors. However, main memory is not amenable to such multiplexing and is often the primary bottleneck in achieving higher degrees of consolidation.

Previous efforts have shown that content-based page sharing provides modest decreases in the memory footprint of VMs running similar operating systems and applications. Our studies show that significant additional gains can be had by leveraging both subpage level sharing (through page patching) and incore memory compression. We build *Difference Engine*, an extension to the Xen VMM, to support each of these—in addition to standard copy-on-write full-page sharing—and demonstrate substantial savings across VMs running disparate workloads (up to 65%). In head-to-head memory-savings comparisons, Difference Engine outperforms VMware ESX server by a factor 1.6–2.5 for heterogeneous workloads. In all cases, the performance overhead of Difference Engine is less than 7%.

1. INTRODUCTION

Virtualization technology has improved dramatically over the past decade to become pervasive within the service-delivery industry. Virtual machines are particularly attractive for server consolidation. Their strong resource and fault isolation guarantees allow multiplexing of hardware among individual services, each with (potentially) distinct software configurations. Anecdotally, individual server machines often run at 5–10% CPU utilization. Operators' reasons are manifold: because of the need to over-provision for peak levels of demand, because fault isolation mandates that individual services run on individual machines, and because many services often run best on a particular operating system configuration. The promise of virtual machine technology for server consolidation is to run multiple services on a single physical machine while still allowing independent configuration and failure isolation.

While physical CPUs are frequently amenable to multiplexing, main memory is not. Many services run comfortably on a machine with 1GB of RAM; multiplexing 10 VMs on that same host, however, would allocate each just 100MB of RAM. Increasing a machine's physical memory is often both difficult and expensive. Incremental upgrades in memory capacity are subject to both the availability of extra slots on the motherboard and the ability to support higher-capacity modules: such upgrades often involve replacing—as opposed to just adding—memory chips. Moreover, not only is high-density memory expensive, it also consumes significant power. Furthermore, as many-core processors become the norm, the bottleneck for VM multiplexing will increasingly be the memory, not the CPU. Finally, both applications and operating systems are becoming more and more resource intensive over time. As a result, commodity operating systems require significant physical memory to avoid frequent paging.

Not surprisingly, researchers and commercial VM software vendors have developed techniques to decrease the memory requirements for virtual machines. Notably, the VMware ESX server implements content-based page sharing, which has been shown to reduce the memory footprint of multiple, homogeneous virtual machines by 10–40%.[19] We find that these values depend greatly on the operating system and configuration of the guest VMs. We are not aware of any previously published sharing figures for mixed-OS ESX deployments. Our evaluation indicates, however, that the benefits of ESX-style page sharing decrease as the heterogeneity of the guest VMs increases, due in large part to the fact that page sharing requires the candidate pages to be *identical*.

The premise of this work is that there are significant additional benefits from sharing at a subpage granularity, i.e., there are many pages that are *nearly* identical. We show that it is possible to efficiently find such similar pages and coalesce them into a much smaller memory footprint. Among the set of similar pages, we are able to store most as *patches* relative to a single baseline page. We also compress

The original version of this paper is entitled "Difference Engine: Harnessing Memory Redundancy in Virtual Machines" and was presented at USENIX OSDI, December 2008. An extended abstract entitled "Difference Engine" appeared in USENIX *;login;* volume 34, number 2.

those pages that are unlikely to be accessed in the near future; an efficient implementation of compression nicely complements page sharing and patching.

In this paper, we present Difference Engine, an extension to the Xen VMM[5] that not only shares identical pages, but also supports subpage sharing and in-memory compression of infrequently accessed pages. Our results show that for a heterogeneous setup (different operating systems hosting different applications), Difference Engine can reduce memory usage by nearly 65%. In head-to-head comparisons against VMware's ESX server running the same workloads, Difference Engine delivers a factor of 1.5 more memory savings for a homogeneous workload and a factor of 1.6–2.5 more memory savings for heterogeneous workloads.

Critically, we demonstrate that these benefits can be obtained without negatively impacting application performance: in our experiments across a variety of workloads, Difference Engine imposes less than 7% overhead. We further show that Difference Engine can leverage improved memory efficiency to increase aggregate system performance by utilizing the free memory to create additional virtual machines in support of a target workload.

2. RELATED WORK

Difference Engine builds upon substantial previous work in page sharing, delta encoding, and memory compression. In each instance, we attempt to leverage existing approaches where appropriate.

2.1. Page sharing

Two common approaches in the literature for finding redundant pages are content-based page sharing, exemplified by VMWare's ESX server,[19] and explicitly tracking page changes to build knowledge of identical pages, exemplified by "transparent page sharing" in Disco.[7] Transparent page sharing can be more efficient, but requires several modifications to the guest OS, in contrast to ESX server and Difference Engine which require no modifications.

To find sharing candidates, both Difference Engine and ESX hash contents of each page and use hash collisions to identify potential duplicates. Once shared, our system can manage page updates in a copy-on-write fashion, as in Disco and ESX server. We build upon earlier work on *flash cloning*[18] of VMs, which allows new VMs to be cloned from an existing VM in milliseconds; as the newly created VM writes to its memory, it is given private copies of the shared pages. An extension by Kloster et al. studied page sharing in Xen[11] and we build upon this experience, adding support for fully virtualized (HVM) guests, integrating the global clock, and improving the overall reliability and performance.

2.2. Delta encoding

Our initial investigations into page similarity were inspired by research in leveraging similarity across files in large file systems. In GLIMPSE,[14] Manber proposed computing Rabin fingerprints over fixed-size blocks at multiple offsets in a file. Similar files will then share some fingerprints. Thus the maximum number of common fingerprints is a strong indicator of similarity. However, in a dynamically evolving virtual memory system, this approach does not scale well since every time a page changes its fingerprints must be recomputed as well. Further, it is inefficient to find the maximal intersecting set from among a large number of candidate pages. Broder adapted Manber's approach to the problem of identifying documents (in this case, Web pages) that are nearly identical using a combination of Rabin fingerprints and sampling based on minimum values under a set of random permutations.[6]

While these techniques can be used to identify similar files, they do not address how to efficiently encode the differences. Douglis and Iyengar explored using Rabin fingerprints and delta encoding to compress similar files in the DERD system,[10] but only considered whole files. Kulkarni et al.[12] extended the DERD scheme to exploit similarity at the block level. Difference Engine also tries to exploit memory redundancy at several different granularities.

2.3. Memory compression

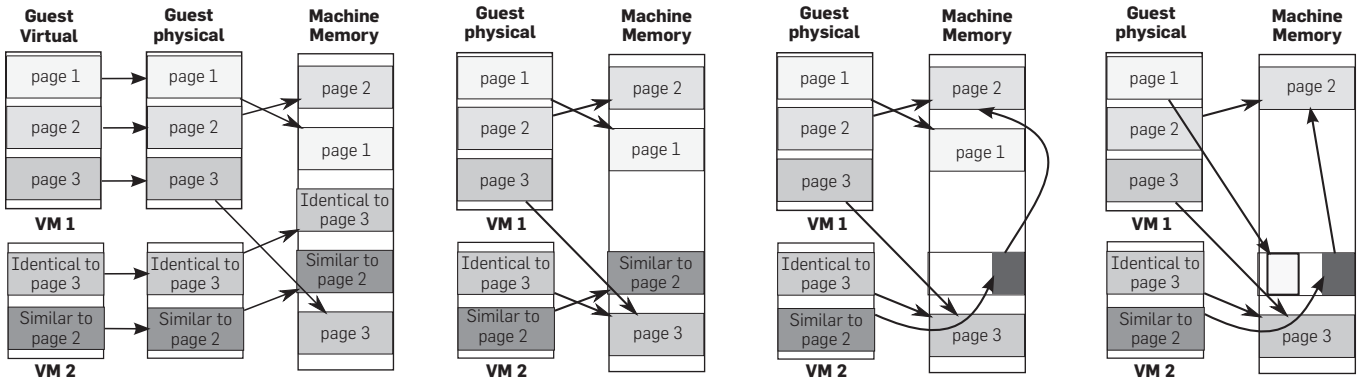
In-memory compression is not a new idea. Douglis et al.[9] implemented memory compression in the Sprite operating system with mixed results. In their experience, memory compression was sometimes beneficial, but at other times the performance overhead outweighed the memory savings. Subsequently, Wilson et al. argued Douglis' mixed results were primarily due to slow hardware.[20] They also developed new compression algorithms (leveraged by Difference Engine) that exploit the inherent structure present in virtual memory, whereas earlier systems used general-purpose compression algorithms. Tuduce et al.[17] implemented a compressed cache for Linux that adaptively manages the amount of physical memory devoted to compressed pages using a simple algorithm shown to be effective across a wide variety of workloads.

3. ARCHITECTURE

Difference Engine uses three distinct mechanisms that work together to realize the benefits of memory sharing, as shown in Figure 1. In this example, two VMs have allocated five pages total, each initially backed by distinct pages in machine memory (Figure 1a). For brevity, we only show how the mapping from guest physical memory to machine memory changes; the guest virtual to guest physical mapping remains unaffected. First, for identical pages across the VMs, we store a single copy and create references that point to the original. In Figure 1b, one page in VM-2 is identical to one in VM-1. For pages that are similar, but not identical, we store a patch against a reference page and discard the redundant copy. In Figure 1c, the second page of VM-2 is stored as a patch to the second page of VM-1. Finally, for pages that are unique and infrequently accessed, we compress them in memory to save space. In Figure 1d, the remaining private page in VM-1 is compressed. The actual machine memory footprint is now less than three pages, down from five pages originally.

In all three cases, efficiency concerns require us to select candidate pages that are unlikely to be accessed in the near future. We employ a global clock that scans memory in the background, identifying pages that have not been recently

Figure 1. The three different memory conservation techniques employed by Difference Engine: page sharing, page patching, and compression. In this example, five physical pages are stored in less than three machine memory pages for a savings of roughly 50%.



used. In addition, reference pages for sharing or patching must be found quickly without introducing performance overhead. Difference Engine uses full-page hashes and hash-based fingerprints to identify good candidates. Finally, we implement a demand paging mechanism that supplements main memory by writing VM pages to disk to support overcommitment, allowing the total memory required for all VMs to temporarily exceed the physical memory capacity.

3.1. Page sharing

Difference Engine's implementation of content-based page sharing is similar to those in earlier systems. We walk through memory looking for identical pages. As we scan memory, we hash each page and index it based on its hash value. Identical pages hash to the same value and a collision indicates that a potential matching page has been found. We perform a byte-by-byte comparison to ensure that the pages are indeed identical before sharing them.

Upon identifying target pages for sharing, we reclaim one of the pages and update the virtual memory to point at the shared copy. Both mappings are marked read-only, so that writes to a shared page cause a page fault that will be trapped by the VMM. The VMM returns a private copy of the shared page to the faulting VM and updates the virtual memory mappings appropriately. If no VM refers to a shared page, the VMM reclaims it and returns it to the free memory pool.

3.2. Patching

Traditionally, the goal of page sharing has been to eliminate redundant copies of *identical* pages. Difference Engine considers further reducing the memory required to store *similar* pages by constructing patches that represent a page as the difference relative to a reference page. To motivate this design decision, we provide an initial study into the potential savings due to subpage sharing, both within and across virtual machines. First, we define the following two heterogeneous workloads, each involving three 512MB virtual machines:

- MIXED-1: Windows XP SP1 hosting RUBiS8; Debian 3.1 compiling the Linux kernel; Slackware 10.2 compiling Vim 7.0 followed by a run of the lmbench benchmark.[15]
- MIXED-2: Windows XP SP1 running Apache 2.2.8 hosting approximately 32,000 static Web pages crawled from Wikipedia, with `httperf` running on a separate machine requesting these pages; Debian 3.1 running the SysBench database benchmark[1] using 10 threads to issue 100,000 requests; Slackware 10.2 running dbench[2] with 10 clients for 6 min followed by a run of the IOZone benchmark.[3]

We designed these workloads to stress the memory-saving mechanisms since opportunities for identical page sharing are reduced. In this first experiment, for a variety of configurations, we suspend the VMs after completing a benchmark, and consider a static snapshot of their memory to determine the number of pages required to store the images using various techniques. Table 1 shows the results of our analysis for the MIXED-1 workload.

The first column breaks down these 393,120 pages into three categories: 149,038 zero pages (i.e., the page contains all zeros), 52,436 sharable pages (the page is not all zeros, and there exists at least one other identical page), and 191,646 unique pages (no other page in memory is exactly the same). The second column shows the number of pages required to store these three categories of pages

Table 1. Effectiveness of page sharing across three 512MB VMs running Windows XP, Debian, and Slackware Linux using 4KB pages.

Pages	Initial	After Sharing	After Patching
Unique	191,646	191,646	
Sharable (non-zero)	52,436	3,577	
Zero	149,038	1	
Total	393,120	195,224	88,422
Reference		50,727	50,727
Patchable		144,497	37,695

using traditional page sharing. Each unique page must be preserved; however, we only need to store one copy of a set of identical pages. Hence, the 52,436 nonunique pages contain only 3,577 distinct pages—implying there are roughly 14 copies of every nonunique page. Furthermore, only one copy of the zero page is needed. In total, the 393,120 original pages can be represented by 195,224 distinct pages—a 50% savings.

The third column depicts the additional savings available if we consider subpage sharing. Using a cut-off of 2KB for the patch size (i.e., we do not create a patch if it will take up more than half a page), we identify 144,497 distinct pages eligible for patching. We store the 50,727 remaining pages as is and use them as reference pages for the patched pages. For each of the similar pages, we compute a patch using Xdelta.[13] The average patch size is 1,070 bytes, allowing them to be stored in 37,695 4KB pages, saving 106,802 pages. In sum, subpage sharing requires only 88,422 pages to store the memory for all VMs instead of 195,224 for fullpage sharing or 393,120 originally—an impressive 77% savings, or almost another 50% over full-page sharing. We note that this was the least savings in our experiments; the savings from patching are even higher in most cases. Further, a significant amount of page sharing actually comes from zero pages and, therefore, depends on their availability. For instance, the same workload when executed on 256MB VMs yields far fewer zero pages. Alternative mechanisms to page sharing become even more important in such cases.

One of the principal complications with subpage sharing is identifying candidate reference pages. Difference Engine uses a parameterized scheme to identify similar pages based upon the hashes of several 64-byte portions of each page. In particular, HashSimilarityDetector(k, s) hashes the contents of ($k \cdot s$) 64-byte blocks at randomly chosen locations on the page, and then groups these hashes together into k groups of s hashes each. We use each group as an index into a hash table. In other words, higher values of s capture *local* similarity while higher k values incorporate *global* similarity. Hence, HashSimilarityDetector(1,1) will choose one block on a page and index that block; pages are considered similar if that block of data matches. HashSimilarityDetector(1,2) combines the hashes from two different locations in the page into one index of length two. HashSimilarityDetector(2,1) instead indexes each page twice: once based on the contents of a first block, and again based on the contents of a second block. Pages that match at least one of the two blocks are chosen as candidates. For each scheme, the number of candidates, c, specifies how many different pages the hash table tracks for each signature. With one candidate, we only store the first page found with each signature; for larger values, we keep multiple pages in the hash table for each index. When trying to build a patch, Difference Engine computes a patch between all matching pages and chooses the best one.

We have evaluated this scheme for a variety of parameter settings on the two workloads described above. For both the workloads, HashSimilarityDetector(2,1) with one candidate does surprisingly well. There is a substantial gain due to hashing two distinct blocks in the page separately, but little additional gain by hashing more blocks. Combining blocks does not help much, at least for these workloads. Furthermore, storing more candidates in each hash bucket also produces little gain. Hence, Difference Engine indexes a page by hashing 64-byte blocks at two fixed locations in the page (chosen at random) and using each hash value as a separate index to store the page in the hash table. To find a candidate similar page, the system computes hashes at the same two locations, looks up those hash table entries, and chooses the better of the (at most) two pages found there.

3.3. Compression

Finally, for pages that are not significantly similar to other pages in memory, we consider compressing them to reduce the memory footprint. Compression is useful only if the compression ratio is reasonably high, and, like patching, if selected pages are accessed infrequently, otherwise the overhead of compression/decompression will outweigh the benefits. We identify candidate pages for compression using a global clock algorithm (Section 4.2), assuming that pages that have not been recently accessed are unlikely to be accessed in the near future.

Difference Engine supports multiple compression algorithms, currently LZO and WKdm as described in Wilson et al.[20]; we invalidate compressed pages in the VM and save them in a dynamically allocated storage area in machine memory. When a VM accesses a compressed page, Difference Engine decompresses the page and returns it to the VM uncompressed. It remains there until it is again considered for compression.

3.4. Paging machine memory

While Difference Engine will deliver some (typically high) level of memory savings, in the worst case all VMs might actually require all of their allocated memory. Setting aside sufficient physical memory to account for this case prevents using the memory saved by Difference Engine to create additional VMs. Not doing so, however, may result in temporarily overshooting the physical memory capacity of the machine and cause a system crash. We therefore require a demand-paging mechanism to supplement main memory by writing pages out to disk in such cases.

A good candidate page for swapping out would likely not be accessed in the near future—the same requirement as compressed/patched pages. In fact, Difference Engine also considers compressed and patched pages as candidates for swapping out. Once the contents of the page are written to disk, the page can be reclaimed. When a VM accesses a swapped out page, Difference Engine fetches it from disk and copies the contents into a newly allocated page that is mapped appropriately in the VM's memory.

Since disk I/O is involved, swapping in/out is an expensive operation. Further, a swapped page is unavailable for sharing or as a reference page for patching. Therefore, swapping should be an infrequent operation. Difference Engine implements the core mechanisms for paging, and leaves policy decisions—such as when and how much to swap—to user space tools.

4. IMPLEMENTATION

We have implemented Difference Engine on top of Xen 3.0.4 in roughly 14,500 lines of code. An additional 20,000 lines come from ports of existing patching and compression algorithms (Xdelta, LZO, WKdm) to run inside Xen.

4.1. Modifications to Xen

Xen and other platforms that support fully virtualized guests use a mechanism called "shadow page tables" to manage guest OS memory.[19] The guest OS has its own copy of the page table that it manages believing that they are the hardware page tables, though in reality it is just a map from the guest's virtual memory to its notion of physical memory (V2P map). In addition, Xen maintains a map from the guest's notion of physical memory to the machine memory (P2M map). The shadow page table is a cache of the results of composing the V2P map with the P2M map, mapping guest virtual memory directly to machine memory.

Difference Engine relies on manipulating P2M maps and the shadow page tables to interpose on page accesses. For simplicity, we do not consider any pages mapped by Domain-0 (the privileged, control domain in Xen), which, among other things, avoids the potential for circular page faults.

4.2. Clock

Difference Engine implements a not-recently-used (NRU) policy[16] to select candidate pages for sharing, patching, compression, and swapping out. On each invocation, the clock scans a portion of the memory, checking and clearing the *referenced* (R) and *modified* (M) bits on pages. Thus, pages with the R/M bits set must have been referenced/modified since the last scan. We ensure that successive scans of memory are separated by at least 4 s in the current implementation to give domains a chance to set the R/M bits on frequently accessed pages. In the presence of multiple VMs, the clock scans a small portion of each VM's memory in turn for fairness. The external API exported by the clock is simple: return a list of pages (of some maximum size) that have not been accessed in some time.

Using the R/M bits, we can annotate each page with its "freshness." By default, we employ the following policy. Recently modified pages are ignored; pages accessed recently but not modified are considered for sharing and to be reference pages for patching, but cannot be patched or compressed themselves; pages not recently accessed can be shared or patched; and pages not accessed for an extended period of time are eligible for everything, including compression and swapping.

4.3. Page sharing

Difference Engine uses the SuperFastHash[4] function to compute digests for each scanned page and inserts them along with the page-frame number into a hash table. Ideally, the hash table should be sized so that it can hold entries for all of physical memory. The hash table is allocated out of Xen's heap space, which is quite limited in size: the code, data, and heap segments in Xen must all fit in a 12MB region of memory. Changing the heap size requires pervasive code changes in Xen, and will likely break the application binary interface (ABI) for some OSes. We therefore restrict the size of the page-sharing hash table so that it can hold entries for only 1/5 of physical memory. Hence Difference Engine processes memory in five passes, as described by Kloster et al.[11] In our test configuration, this partitioning results in a 1.76MB hash table. We divide the space of hash function values into five intervals, and only insert a page into the table if its hash value falls into the current interval. A complete cycle of five passes covering all the hash value intervals is required to identify all identical pages.

4.4. Page-similarity detection

The goal of the page-similarity component is to find pairs of pages with similar content, and, hence, make candidates for patching. We implement a simple strategy for finding similar pages based on hashing short blocks within a page, as described in Section 3.2. Specifically, we use the HashSimilarityDetector(2,1) described there, which hashes short data blocks from two locations on each page, and indexes the page at each of those two locations in a separate page-similarity hash table, distinct from the page-sharing hash table described above. We use the 1-candidate variation, where at most one page is indexed for each block hash value.

Recall that the clock makes a complete scan through memory in five passes. The page-sharing hash table is cleared after each pass, since only pages *within* a pass are considered for sharing. However, two similar pages may appear in different passes if their hash values fall in different intervals. Since we want to only consider pages that have not been shared in a full cycle for patching, the page-similarity hash table is *not* cleared on every pass. This approach also increases the chances of finding better candidate pages to act as the reference for a patch.

4.5. Compression

Compression operates similarly to patching—in both cases the goal is to replace a page with a shorter representation of the same data. The primary difference is that patching makes use of a reference page, while a compressed representation is self contained. There is one important interaction between compression and patching: once we compress a page, the page can no longer be used as a reference for a later patched page. A naive implementation that compresses all nonidentical pages as it goes along will almost entirely prevent page patches from being built. Compression of a page should be postponed at least until all pages have been checked for similarity against it. A complete cycle of a page sharing scan will identify similar pages, so a sufficient condition for compression is that *no page should be compressed until a complete cycle of the page sharing code finishes*.

4.6. Paging machine memory

Recall that any memory freed by Difference Engine cannot be used reliably without supplementing main memory with secondary storage. That is, when the total allocated memory of all VMs exceeds the system memory capacity, some pages will have to be swapped to disk.

The Xen VMM does not perform any I/O (delegating all I/O to Domain-0) and is not aware of any devices. Thus, it is not possible to build swap support directly in the VMM. Further, since Difference Engine supports unmodified OSes, we cannot expect any support from the guest OS. Hence, we implement a single swap daemon (`swapd`) running as a user process in Domain-0 to manage swap space. For each VM in the system, `swapd` creates a separate thread to handle swap-in requests.

To swap out a page, `swapd` makes a hypercall into Xen, where a victim page is chosen by invoking the global clock. If the victim is a compressed or patched page, we first reconstruct it. We pause the VM that owns the page and copy the contents of the page to a page in Domain-0's address space (supplied by `swapd`). Next, we remove all entries pointing to the victim page in the P2M and M2P maps, and in the shadow page tables. We then mark the page as swapped out in the corresponding page table entry. Meanwhile, `swapd` writes the page contents to the swap file and inserts the corresponding byte offset in a hash table keyed by <Domain ID, guest page-frame number>. Finally, we free the page, return it to the domain heap, and reschedule the VM.

When a VM tries to access a swapped page, it incurs a page fault and traps into Xen. We pause the VM and allocate a fresh page to hold the swapped in data. We populate the P2M and M2P maps appropriately to accommodate the new page. Xen dispatches a swap-in request to `swapd` containing the domain ID and the faulting page-frame number. The handler thread for the faulting domain in `swapd` receives the request and fetches the location of the page in the swap file from the hash table. It then copies the page contents into the newly allocated page frame within Xen via another hypercall. At this point, `swapd` notifies Xen, and Xen restarts the VM at the faulting instruction.

5. EVALUATION

We first present micro-benchmarks to evaluate the cost of individual operations, the performance of the global clock and the behavior of each of the three mechanisms in isolation. Next, we evaluate whole system performance: for a range of workloads, we measure memory savings and the impact on application performance. We present head-to-head comparisons with the VMware ESX server. Finally, we demonstrate how our memory savings can be used to boost the aggregate system performance. Unless otherwise mentioned, all experiments are run on dual-processor, dual-core 2.33-GHz Intel Xeon machines and the page size is 4KB.

5.1. Cost of individual operations

Before quantifying the memory savings provided by Difference Engine, we measure the overhead of various functions involved. Table 2 shows the overhead imposed by the major Difference Engine operations. As expected, collapsing identical pages into a copy-on-write shared page (`share_page`) and recreating private copies (`cow_break`) are relatively cheap operations, taking approximately 6 and 25 µs, respectively. Perhaps more surprising, however, is that compressing a page on our hardware is fast, requiring slightly less than 30 µs on average. Patching, on the other hand, is almost an order of magnitude slower: creating a patch (`patch_page`) takes over 300 µs. This time is primarily due to the overhead of finding a good candidate base page and constructing the patch. Both decompressing a page and reconstructing a patched page are also fairly fast, taking 10 and 18 µs respectively.

Table 2. CPU overhead of different functions.

Function	Mean execution time (µs)
`share_pages`	6.2
`cow_break`	25.1
`compress_page`	29.7
`uncompress`	10.4
`patch_page`	338.1
`unpatch`	18.6
`swap_out_page`	48.9
`swap_in_page`	7151.6

Swapping out takes approximately 50 µs. However, this does *not* include the time to actually write the page to disk. This is intentional: once the page contents have been copied to user space, they *are* immediately available for being swapped in; and the actual write to the disk might be delayed because of file system and OS buffering in Domain-0. Swapping in, on the other hand, is the most expensive operation, taking approximately 7 ms. There are a few caveats, however. First, swapping in is an asynchronous operation and might be affected by several factors, including process scheduling within Domain-0; it is *not* a tight bound. Second, swapping in might require reading the page from disk, and the seek time will depend on the size of the swap file, among other things.

5.2. Real-world applications

We now present the performance of Difference Engine on a variety of workloads. We seek to answer two questions. First, how effective are the memory-saving mechanisms at reducing memory usage for real-world applications? Second, what is the impact of those memory-sharing mechanisms on system performance? Since the degree of possible sharing depends on the software configuration, we consider several different cases of application mixes.

To put our numbers in perspective, we conduct head-to-head comparisons with VMware ESX server for three different workload mixes. We run ESX Server 3.0.1 build 32,039 on a Dell PowerEdge 1950 system. Note that even though this system has two 2.3-GHz Intel Xeon processors, our VMware license limits our usage to a single CPU. We therefore restrict Xen (and, hence, Difference Engine) to use a single CPU for fairness. We also ensure that the OS images used with ESX match those used with Xen, especially the file system and disk layout. Note that we are only concerned with the effectiveness of the memory sharing mechanisms—not in comparing the application performance across the two hypervisors. In an effort to compare the performance of Difference Engine to ESX in its most aggressive configuration, we configure both to scan 10,000 pages/s, the highest rate allowed by ESX.[a]

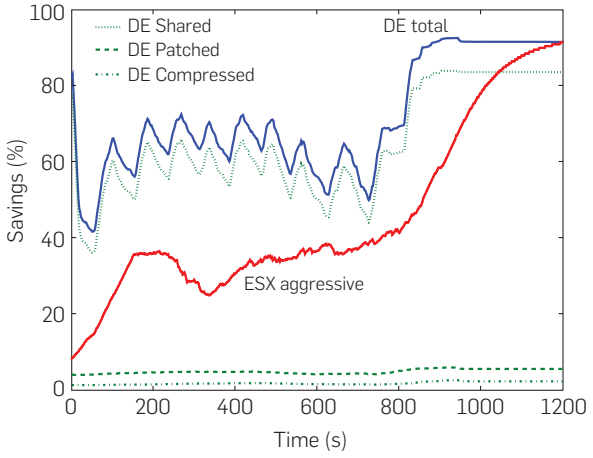
Base Scenario—Homogeneous VMs: In our first set of benchmarks, we test the base scenario where all VMs on a machine run the same OS and applications. This scenario is common in cluster-based systems where several services are replicated to provide fault tolerance or load balancing. Our expectation is that significant memory savings are available and that most of the savings will come from page sharing.

On a machine running standard Xen, we start from 1 to 6 VMs, each with 256MB of memory and running RUBiS[8]—an e-commerce application designed to evaluate application server performance—on Debian 3.1. We use the PHP implementation of RUBiS; each instance consists of a Web server (Apache) and a database server (MySQL). Two distinct client machines generate the workload, each running the standard RUBiS workload generator simulating 100 user sessions. The benchmark runs for roughly 20 min. The workload generator reports several metrics at the end of the benchmark, in particular the average response time and the total number of requests served.

We then run the same set of VMs with Difference Engine enabled. Both the total number of requests and the average response time remain unaffected while Difference Engine delivers 65%–75% memory savings. In this case, the bulk of memory savings comes from page sharing. Recall that Difference Engine tries to share as many pages as it can before considering pages for patching and compression, so sharing is expected to be the largest contributor in most cases, particularly in homogeneous workloads.

We next compare Difference Engine performance with the VMware ESX server. We set up four 512MB virtual machines running Debian 3.1. Each VM executes dbench[2] for 10 min followed by a stabilization period of 20 min. Figure 2 shows the amount of memory saved as a function of time. First, note that *eventually* both ESX and Difference Engine reclaim roughly the same amount of memory (the graph for ESX plateaus beyond 1,200 s). However, *while* dbench is executing, Difference Engine delivers approximately 1.5 times the memory savings achieved by ESX. As before, the bulk of Difference Engine savings come from page sharing for the homogeneous workload case.

Figure 2. Four identical VMs execute dbench. Both Difference Engine and ESX eventually yield similar savings, but DE extracts more savings *while* the benchmark is in progress.

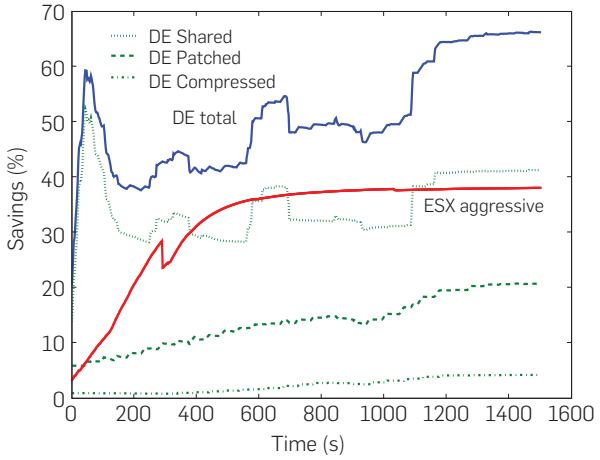


Heterogeneous OS and Applications: Given the increasing trend toward virtualization, both on the desktop and in the data center, we envision that a single physical machine will host significantly different types of operating systems and workloads. While smarter VM placement and scheduling will mitigate some of these differences, there will still be a diverse and heterogeneous mix of applications and environments, underscoring the need for mechanisms other than page sharing. We now examine the utility of Difference Engine in such scenarios, and demonstrate that significant additional memory savings result from employing patching and compression in these settings.

Figures 3 and 4 show the memory savings as a function of time for the two heterogeneous workloads—MIXED-1 and MIXED-2 described in Section 3.2. We make the following observations. First, in steady state, Difference Engine delivers a factor of 1.6–2.5 more memory savings than ESX. For instance, for the MIXED-2 workload, Difference Engine could host the three VMs allocated 512MB of physical memory each in approximately 760MB of machine memory; ESX would require roughly 1,100MB of machine memory. The remaining, significant, savings come from patching and compression. And these savings come at a small cost. Table 3 summarizes the performance of the three benchmarks in the MIXED-1 workload. The baseline configuration is regular Xen without Difference Engine. In all cases, performance overhead of Difference Engine is within 7% of the baseline. For the same workload, we find that performance under ESX with aggressive page sharing is also within 5% of the ESX baseline with no page sharing.

Increasing Aggregate System Performance: Difference Engine goes to great lengths to reclaim memory in a system, but eventually this extra memory needs to actually get used in a productive

Figure 3. Memory savings for MIXED-1. Difference Engine saves up to 45% more memory than ESX.



[a] After initial publication of our results, we were informed by VMware that this version of ESX silently limits the effective page-sharing rate to a maximum of 450 pages/sec per vm regardless of the configured scan rate.

Figure 4. Memory savings for Mixed-2. Difference Engine saves almost twice as much memory as ESX.

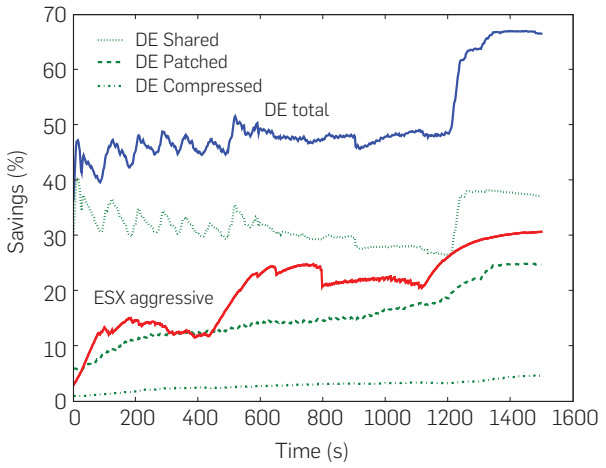


Figure 5. Up to a limit, Difference Engine can help increase aggregate system performance by spreading the load across extra VMs.

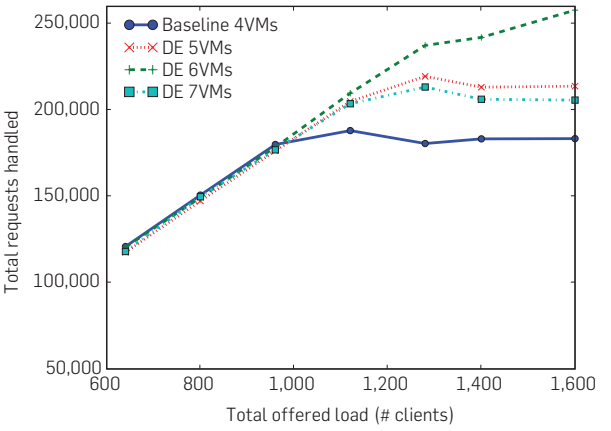


Table 3. Application performance under Difference Engine for the heterogeneous workload Mixed-1 is within 7% of the baseline.

	Kernel Compile (s)	Vim compile, lmbench (s)	RUBiS requests	RUBiS response time(ms)
Baseline	670	620	3149	1280
DE	710	702	3130	1268

manner. One can certainly use the saved memory to create more VMs, but does that increase the aggregate system performance?

To answer this question, we created four VMs with 650MB of RAM each on a physical machine with 2.8GB of free memory (excluding memory allocated to Domain-0). For the baseline (without Difference Engine), Xen allocates memory statically. Upon creating all the VMs, there is clearly not enough memory left to create another VM of the same configuration. Each VM hosts a RUBiS instance. For this experiment, we used the Java Servlets implementation of RUBiS. There are two distinct client machines per VM to act as workload generators.

The goal is to increase the load on the system to saturation. The solid line in Figure 5 shows the total requests served for the baseline, with the total offered load marked on the X-axis. Beyond 960 clients, the total number of requests served plateaus at around 180,000 while the average response time (not shown) increases sharply. Upon investigation, we find that for higher loads all of the VMs have more than 95% memory utilization and some VMs actually start swapping to disk (within the guest OS). Using fewer VMs with more memory (e.g., 2 VMs with 1.2GB RAM each) did not improve the baseline performance for this workload.

Next, we repeat the same experiment with Difference Engine, except this time we utilize reclaimed memory to create additional VMs. As a result, for each data point on the X-axis, the per VM load decreases, while the aggregate offered load remains the same. We expect that since each VM individually has lower load compared to the baseline, the system will deliver better aggregate performance. The remaining lines show the performance with up to three extra VMs. Clearly, Difference Engine enables higher aggregate performance compared to the baseline. However, beyond a certain point (two additional VMs in this case), the overhead of managing the extra VMs begins to offset the performance benefits: Difference Engine has to manage 4.5GB of memory on a system with 2.8GB of RAM to support seven VMs. In each case, beyond 1,400 clients, the VM's working set becomes large enough to invoke the paging mechanism: we observe between 5,000 pages (for one extra VM) to around 20,000 pages (for three extra VMs) being swapped out, of which roughly a fourth get swapped back in.

6. CONCLUSION

One of the primary bottlenecks to higher degrees of virtual machine multiplexing is main memory. Earlier work shows that substantial memory savings are available from harvesting identical pages across virtual machines when running homogeneous workloads. The premise of this work is that there are significant additional memory savings available from locating and patching similar pages and in-memory page compression. We present the design and evaluation of Difference Engine to demonstrate the potential memory savings available from leveraging a combination of whole page sharing, page patching, and compression. Our performance evaluation shows that Difference Engine delivers an additional factor of 1.6–2.5 more memory savings than VMware ESX server for a variety of workloads, with minimal performance overhead. Difference Engine mechanisms might also be leveraged to improve single OS memory management; we leave such exploration to future work.

Acknowledgments

In the course of the project, we received invaluable assistance from a number of people at VMware. We would like to thank Carl Waldspurger, Jennifer Anderson, and Hemant Gaidhani, and the Performance Benchmark group for feedback and discussions on the performance of ESX server. Also, special thanks are owed to Kiran Tati for assisting

with ESX setup and monitoring and to Emil Sit for providing insightful feedback on the paper. Finally, we would like to thank Michael Mitzenmacher for his assistance with min-wise hashing, our OSDI shepherd Fred Douglis for his insightful feedback and support, Rick Farrow at *;login;* and the anonymous OSDI '08 reviewers for their valuable comments. This work was supported in part by NSF CSR-PDOS Grant No. CNS-0615392, the UCSD Center for Networked Systems (CNS), and UC Discovery Grant 07–10237. Vrable was supported in part by an NSF Graduate Research Fellowship.

References

1. http://sysbench.sourceforge.net/.
2. http://samba.org/ftp/tridge/dbench/.
3. http://www.iozone.org/.
4. http://www.azillionmonkeys.com/qed/hash.html.
5. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003).
6. Broder, A.Z. Identifying and filtering near-duplicate documents. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching* (2000).
7. Bugnion, E., Devine, S., Rosenblum, M. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating System Principles* (1997).
8. Cecchet, E., Marguerite, J., Zwaenepoel, W. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM Conference on Object-oriented Programming, Systems, Languages, and Applications* (2002).
9. Douglis, F. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the USENIX Winter Technical Conference* (1993).
10. Douglis, F., Iyengar, A. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference* (2003).
11. Kloster, J.F., Kristensen, J., Mejlholm, A. On the feasibility of memory sharing. Master's thesis, Aalborg University (2006).
12. Kulkarni, P., Douglis, F., Lavoie, J., Tracey, J.M. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference* (2004).
13. MacDonald, J. xdelta. http://www.xdelta.org/.
14. Manber, U., Wu, S. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter Technical Conference* (1994).
15. McVoy, L., Staelin, C. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference* (1996).
16. Tanenbaum, A.S. *Modern Operating Systems*. Prentice Hall (2007).
17. Tuduce, I.C., Gross, T. Adaptive main memory compression. In *Proceedings of the USENIX Annual Technical Conference* (2005).
18. Vrable, M., Ma, J., Chen, J., Moore, D., VandeKieft, E., Snoeren, A.C., Voelker, G.M., Savage, S. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating System Principles* (2005).
19. Waldspurger, C.A. Memory resource management in VMware ESX server. In *Proceedings of the 5th ACM/USENIX Symposium on Operating System Design and Implementation* (2002).
20. Wilson, P.R., Kaplan, S.F., Smaragdakis, Y. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference* (1999).

Diwaker Gupta (diwaker@asterdata.com), University of California, San Diego, Currently at Aster Data.

Sangmin Lee (sangmin@cs.utexas.edu), University of California, San Diego, Currently at UT Austin.

Michael Vrable (mvrable@cs.ucsd.edu), University of California, San Diego.

Stefan Savage (savage@cs.ucsd.edu), University of California, San Diego.

Alex C. Snoeren (snoeren@cs.ucsd.edu), University of California, San Diego.

George Varghese (varghese@cs.ucsd.edu), University of California, San Diego.

Geoffrey M. Voelker (voelker@cs.ucsd.edu), University of California, San Diego.

Amin Vahdat (vahdat@cs.ucsd.edu), University of California, San Diego.