Memory Buddies: Exploiting Page Sharing for Server Consolidation in Virtualized Data Centers

Timothy Wood, Gabriel Tarasuk-Levin†, Jim Cipar*, Peter Desnoyers
Prashant Shenoy, Mark Corner and Emery Berger
Univ. of Massachusetts Amherst †Hampshire College *Carnegie Mellon University

ABSTRACT

Modern data centers increasingly employ virtualization in order to maximize resource utilization while reducing costs. With the advent of multi-core processors, memory has become the key limiting resource when running multiple virtual machines on a given server. In this paper, we argue for exploiting page sharing to significantly reduce aggregate memory requirements of collocated VMs and increase the number of VMs that can be housed on a given data center. We present a memory tracer and a novel memory fingerprinting technique—based on Bloom Filters— to concisely and efficiently capture the memory contents of each VM. Our server consolidation algorithm employs a fast, scalable fingerprint comparison technique to efficiently determine which data center servers offer the maximum sharing potential for a given VM, and consolidates these VMs onto these servers via live migration. We also present a hotspot mitigation technique to correct the negative impacts of unanticipated loss of page sharing and swapping. We have implemented a prototype of our Memory Buddies system in the VMware ESX server. An experimental evaluation of our prototype using a mix of enterprise and ecommerce applications demonstrates that exploiting page sharing enables our system to increase the effective capacity of a data center by over 30%, while imposing a CPU and network overhead of less than 0.5% per server and scaling to data centers with up to a thousand servers.

1. INTRODUCTION

Data centers—server farms that run networked applications—have become popular in a variety of domains such as web hosting, enterprise applications, and e-commerce sites. Modern data centers are increasingly employing a virtualized architecture where applications run inside virtual servers that are constructed using virtual machines, and one or more virtual servers are mapped onto each physical server in the data center. Running applications inside virtual servers as opposed to running them directly on physical servers provides a number of benefits. First, it enables server consolidation since multiple applications can be housed on a single physical server while still providing the illusion of running each one on an independent (virtual) server. This reduces the number of physical servers required to house a given set of applications and also provides better multiplexing of data center resources across applications. Second, consolidating applications on a smaller set of physical servers enables spare servers to be powered off until needed, which reduces the energy costs of operating the data center. Third, it enables legacy applications that require legacy operating systems to be migrated to virtual machines without a need to port the application to newer operating systems.

With the advent of multi-core processors, it is now feasible to run up to a few tens of virtual machines on a single physical server. Since memory is a limited and expensive server resource, it can become the limiting factor when trying to place multiple virtual machines onto a single server. In this paper, we argue for exploiting page sharing between virtual machines to maximize the number of VMs that can be placed on a server during initial placement or while consolidating servers. If two virtual machines on a server have identical pages in memory, they can share a single copy of the duplicate page, thereby reducing their total memory needs. Thus, exploiting page sharing can reduce the aggregate memory footprint of virtual servers that reside on a host and enable more VMs to be packed on to a given set of physical servers. Page sharing mechanisms are already available in virtual machines such as VMware and Xen, and have been shown to provide significant memory savings [24]. However, determining *which* VMs to colocate on each physical server so as to best exploit page sharing is a non-trivial problem. First, although similar to bin-packing, there is no known analog of this "packing with sharing" problem in the theory literature; even without sharing, the consolidation problem is an instance of bin-packing, which is NP-complete. A further difficulty is that a simple estimate of the sharing potential between two virtual machines may be inaccurate in the long term. A naive estimate may expect certain pages to be shared, when in reality they may be swapped out after the VMs are co-located on the same host.

In this paper, we present *Memory Buddies*, a sys-

tem that enables server consolidation by aggressively exploiting page sharing benefits. Our system consolidates virtual machines from underloaded servers by colocating them with other VMs with the greatest page sharing potential; doing so reduces the aggregate number of servers needed to house a given set of applications and saves on hardware and energy costs. At the heart of the Memory Buddies system is a *memory fingerprinting* technique that can concisely capture the memory contents of a particular VM as well as the aggregate memory contents of all VMs resident on a physical server. Memory fingerprinting is implemented using (i) a *memory tracer* that traces the VM's memory activity to rank the importance of each page and produce accurate long term estimate of sharing, (ii) a *bloom filter* that uses these tracer statistics to produce a concise fingerprint of the VM's memory contents. Our consolidation algorithm employs a fast fingerprint comparison technique that can quickly compare a VM's fingerprint with those of physical servers and identify servers with the greatest page sharing potential for that VM. Both our fingerprint estimation and fingerprint comparison techniques are designed to scale well to data centers with hundreds or thousands of hosts, while imposing minimal memory tracing overhead. Our system also implements a memory hotspot mitigation algorithm to reduce the impact of memory swapping caused by loss of page sharing or major application phase changes.

We have implemented our techniques using the VMware ESX server on a prototype Linux data center. An experimental evaluation of our prototype has shown that exploiting page sharing for placement in large data centers can increase the number of virtual machines which can be hosted on a data center by up to 30%. We have demonstarted that our Bloom Filter based fingerprinting mechanism can accurately and efficiently determine the amount of sharing between virtual machines and that the low overhead of our system allows it to scale well to large data centers.

The rest of this paper is structured as follows. Section 2 present some background and a problem formulation. Section 3 presents our memory fingerprinting technique. Sections 4 and 5 present our server consolidation and hotspot mitigation algorithms. Section 6 presents implementation details and Section 7 our evaluation results. We present related work in Section 8 and our conclusions in Section 9.

2. BACKGROUND AND SYSTEM OVERVIEW

Our work assumes a data center comprising a large cluster of possibly heterogeneous servers. The hardware configuration of each server—the processor, memory and network characteristics—is assumed to be known *a priori*. Each physical server runs a hypervisor (also referred to as a virtual machine monitor) and one or more virtual machines. Each virtual machine runs an application or a component of an application, and is allocated a certain slice of the underlying physical server. The slice, which determines the fraction of the RAM as well as processor and network bandwidth allocated to the VM, is determined based on the SLA for that application. All storage is assumed to be on a network file system or a storage area network, which eliminates the need to move disk state if the VM is migrated to another physical server [5].

The underlying virtual machine monitor is assumed to implement a page sharing mechanism, which detects duplicate memory pages in resident VMs and uses a single physical page that is shared by all such VMs (if a shared page is subsequently modified by one of the VMs, it must be unshared similar to a copy-on-write mechanism) [24]. Thus, if VM_1 contains M_1 pages, and VM_2 contains M_2 pages, and S of these pages are common across the two VMs, then page sharing can reduce the total memory footprint of two VMs to $M_1 + M_2 - S$ from $M_1 + M_2$. The freed up memory can be used to house other VMs, and enables a larger set of VMs to be placed on a given cluster. In this work, we assume the VMware ESX server, which implements such a page sharing mechanism [24].

Problem formulation: Assuming the above scenario, the server consolidation problem can be formulated as follows. Consider a data center that needs to consolidate one or more physical servers. A physical server becomes a candidate for consolidation if its load stays below a low threshold for an extended period of time;[1] consolidating its VMs on other servers allows the server to be powered down.

For each VM resident on such a server, the consolidation algorithm should to determine a new host such that (i) the new server has sufficient unused resources to house the incoming VM, and (ii) page sharing is maximized. By placing the VM onto a host that maximizes the sharing of that VM's pages, the aggregate memory footprint of the VM can be reduced, enabling greater consolidation. The same strategy can also be used for initial placement of a new VM—the new VM is placed onto a server with the maximum sharing potential, causing the VM to consume less physical memory than otherwise. Once a new target machine has been determined for each VM on a consolidation candidate, *live migration* can be used to actually move the VM to the new server without incurring application down-time [17, 5].

Typically the best sharing benefits will be accrued for duplicate pages that are both active and remain unmodified in memory for extended periods of time. Although the consolidation algorithm should only consider such

[1]A server may also become a consolidation candidate if it needs to be retired from service.

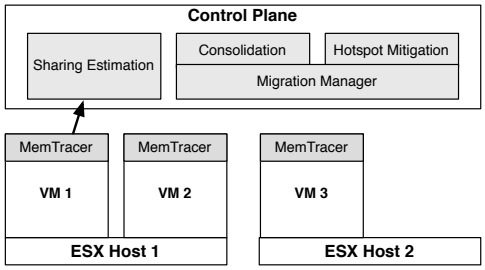


Figure 1: Memory Buddies System Architecture.

pages while estimating sharing potential, it is inevitable that future VM behavior will occasionally deviate from these estimates. This may occur, for instance, if the VM terminates the current application and begins executing a new one, or if the current application sees a major phase change in its behavior. In such a scenario, previously shared pages may become unshared, increasing the memory pressure on the physical server. If the increased memory pressure results in swapping, then application performance can be significantly impacted, resulting in a memory hotspot. To mitigate the impact of such hotspots, we require our consolidation algorithm to incorporate memory hotspot mitigation techniques. Such a technique can take corrective action by offloading one or more VMs to other servers to reduce the memory pressure. Such hotspot mitigation is essential for system stability, especially since aggressive consolidation will occasionally result in poor decisions. Memory hotspot mitigation and consolidation have opposing goals—whereas the latter aggregates load from underloaded servers onto a smaller set of servers, the former dissipates load from overloaded servers onto a set of less loaded servers.

System Overview. Low-level page sharing mechanisms only detect and share duplicate pages belonging to resident VMs—they do not address the problem of *which* VMs to co-locate on a host to maximize sharing. Our *Memory Buddies* system detects sharing potential between virtual machines throughout a cluster of hosts, and then uses the low-level sharing mechanisms to realize these benefits.

Our system, which is depicted in Figure 1 consists of a *nucleus*, which runs on each server, and a *control plane*, which runs on a distinguished control server. Each nucleus runs our memory tracer to gather memory statistics for each resident VM. The tracer tracks accesses to pages within a virtual machine in order to determine what pages are being actively used. It then generates a memory fingerprint for each VM, which is a concise representation of the VM's memory contents, as well as an aggregate fingerprint for the server. The active memory statistics and the various fingerprints are reported to the control plane.

The control plane is responsible for server consolidation and hotspot mitigation. Upon identifying a server for consolidation, it employs a fast fingerprint comparison algorithm to determine a new home for each VM and triggers VM migrations to those hosts. Similarly upon detecting a memory hotspot, it migrates one or more resident VMs to other hosts to relieve memory pressure on that server.

The following sections describe the memory fingerprinting and control plane algorithms in detail.

3. MEMORY FINGERPRINTING

The nucleus runs on each physical server and its goal is to compute a memory fingerprint for each resident VM as well as an aggregate memory fingerprint for the physical server. Ideally the nucleus can be implemented at the VM and the hypervisor-layers to track memory accesses by the guest operating system and its applications in order to generate a fingerprint based on these accesses. However, since VMware is a commercial product, we lack source code access to make modifications to the virtual machine and the hypervisor; consequently we choose to implement this same functionality inside the guest operating system. In open-source platforms such as Xen, it is easy to implement this functionality at the VM level without a need to modify guest operating systems.[2] Thus, our current nucleus runs inside each VM and is implemented as a kernel module and a user-space daemon.

Determining page sharing potential involves computing how many pages are common to two (or more) virtual machines. Our experimental section demonstrates that a brute force comparison of two VMs to determine if they have duplicate memory pages is very expensive, and a more efficient method must be used to (i) succinctly capture the memory contents of each VM, and (ii) compare the memory contents of two or more VMs to determine how many duplicates are present. Further, even if a page is duplicated across VMs, it may not yield sharing benefits—sharing benefits will accrue only if the page is not modified for a long period of time and if it remains active in memory for an extended period. Thus, we must only consider resident pages that are unlikely to be modified when computing the sharing potential.

Our memory tracer addresses the issue of tracking resident pages, while our memory fingerprinting generates a concise representation of the memory contents of these pages. We describe each component in turn.

[2]Our initial efforts had focused on the Xen platform. However, Xen's page sharing implementation is experimental and not compatible with its live migration mechanism; since our work requires both, we chose to switch to the VMware ESX server.

3.1 Memory Tracer

The first step in determining the set of resident pages is to gather a trace of the memory activity of the guest system to produce an ordered list of page accesses. This is generated by periodically sampling the accessed bits for pages in physical memory to see if they have been recently used. The sampler picks a page at random and checks its accessed bit. If the bit is set, then it appends an identifier for the page to the trace and clears the bit. The identifier is dependent on the address space the page belongs to and its offset. By using these as the identifier, a virtual page which is swapped out, and swapped back in to a different physical address will have the same identifier. This allows the stack algorithm, described below, to track the activity of virtual pages, and assess when a guest would benefit from an increase in memory allocation.

In addition to tracing the identifiers for the accessed pages, the tracer also records a hash of their contents using the SuperFast Hash algorithm [9]. To avoid excessive CPU overhead, these hashes are cached so that they do not need to be computed for every page access. Thus the hashes may not be consistent with the page's true contents. Every time a page access is detected, the system must decide whether or not to recompute the hash. Ideally the hash would be recomputed only if the page had been modified, or if the hash had never been previously computed. Unfortunately the Linux kernel does not have an API for checking the dirty bit of every mapping of a virtual page. To prevent the need for extensive kernel modifications, we chose a simpler route: if the page has never been hashed before, then it is always hashed, otherwise it is hashed with some probability, currently 0.01. This means that the hash for a page may not be current, but has probably been updated within the last 99 samples to that page. This is a justifiable approach because pages which do not change frequently are likely to have a correct hash. Pages which do change frequently are likely to have a stale hash, but these pages are unlikely to be involved in sharing, so an incorrect hash will cause few problems.

The kernel component of the memory tracer generates a stream of page access data which is exported to the guest's user space via a character device called `/dev/memtracer`. The user space component reads this file and uses Mattson's Stack Algorithm [16] to determine the set of active pages. The stack algorithm simulates the behavior of an infinitely sized LRU queue acting on the memory accesses indicated in the trace. It does this using two key data structures: a sorted list of pages and an array of integers A. The list of pages stores the fictional LRU queue, and is structured as a modified Red-Black tree [6] which is sorted by LRU position rather than page ID. This allows the tracer to determine the location of an element in the queue in $O(logn)$ time by walking up the tree and observing the number of left-hand children for parent. Indexing into the queue is done via a hash table which maps page IDs to tree elements. The array of integers is indexed by LRU position, and each element represents the number of times an access occurred to that position in the queue.

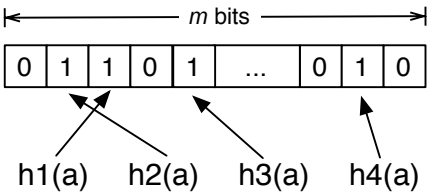


Figure 2: Bloom Filter with four hash functions.

Thus, the memory tracer generates a list of resident memory pages, in LRU order, and a list of hashes that capture the contents of each page. Our implementation also allows the system to request a list of only the active pages–a subset of the resident pages selected based on their LRU position. Currently, we base each virtual machine's fingerprint on the full set of resident pages, however we are investigating using the LRU list to build a Miss Ratio Curve which can be used to more precisely determine the amount of memory required by a virtual machine [3]. This would allow us to safely reduce the memory allocation being given virtual machines not actively using their full allocation and provide performance based SLAs.

3.2 Fingerprint Generation

Content based page sharing implementations for both Xen and VMWare utilize page hashes–such as those generated by the memory tracer—in order to determine if a memory page can be shared with another virtual machine [24, 12]. Maintaining hash lists of the pages being used by each virtual machine provide an exact knowledge of the virtual machine's memory. However, the storage and network requirements of maintaining this information and periodically transmitting it to the control plane may be too high—creating a 32 bit hash for each 4KB page results in 1MB of hash data per 1GB of memory, resulting in a high overhead for a data center with hundreds of machines.

To reduce this overhead, we must generate a succinct representation of a VM's memory contents. We refer to such a summary as the VM's *memory fingerprint.* Our Memory Buddies system uses bloom filters to generate a fingerprint of each virtual machine's memory image. A Bloom Filter is essentially a lookup function implemented using hash functions, and can be used to insert and lookup elements [1, 14]. As shown in Figure 2, a Bloom Filter consists of an m-bit vector and a set of k hash functions $H = h_1, h_2, h_3, ..., h_k$ ($k = 4$ in the fig-

ure). Initially, all bits of the vector are set to zero. To insert an element a, the bits corresponding to positions $h_1(a), h_2(a), ..., h_k(a)$ are set to 1. Observe that a false positive can result if the bit positions for an object are set by other objects that hash to these positions. The probability of false positives (errors) depends on the size of the vector m and the number of hash functions used k. The probability of false positives (errors) assuming no deletions in the bit vector or with deletions permitted on a vector with integer counters, depends on the size of the vector m and the number of hash functions used k. If the number of elements stored in the vector are n, the error probability 'p_e' is given by,

$$p_e = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k \quad (1)$$

Thus, given n, the number of pages belonging to a VM, proper choice of the size of the bit vector m and the number of hash functions k can yield a sufficiently small error probability. For example, $n = 256K$ for a 1 GB VM with 4KB pages, and suppose a $m = 512KB$ Bloom Filter is desired. Setting $k = 11$ in equation 1 produces a false positive rate of 0.046%. This selection of k produces the minimum error rate for the given values of m and n.

Our system uses the hash list generated by the memory tracer to construct the Bloom Filter—the page hashes generated are treated as keys and are inserted into Bloom Filters to generated a compact representation of the memory of each virtual machine. Observe that if the same page is present in two different VMs, the corresponding hash (key) will be inserted into both Bloom Filters, and the same set of bits will be turned on in both fingerprints.

Given the individual fingerprints of all VMs resident on a server, it is easy to generate a composite fingerprint for a physical server, which represents the aggregate memory contents of all resident pages on that server. This is achieved by simplying OR'ing together the bit vectors from each virtual machine's Bloom Filter:

$$BF_{server} = BF_{vm_1} + BF_{vm_2} + \ldots + BF_{vm_k}$$

This is equivalent to combining the various hash lists and inserting them into a new Bloom Filter—since the OR operation will set all bits that would be set by inserting keys from the combined hash list.

4. SHARING-AWARE CONSOLIDATION

The VM and server fingerprints are periodically computed and transmitted to the control plane by each nucleus; the control plane thus has a system-wide view of the fingerprints of all VMs and servers in the data center. The control plane implements a consolidation algorithm that uses this system-wide knowledge to identify servers with the greatest page sharing potential for each VM that needs consolidation. This section first describes how bloom-filter-based fingerprints can be employed to estimate page sharing potential, and then describes the consolidation algorithm employed by the control plane.

4.1 Fast Fingerprint Comparison

To estimate page sharing potential, we need an efficient means to compare their memory contents of two or more virtual machines. An intersection of the hash lists of the VMs can provide accurate information about which pages are duplicates, but is too expensive; instead we must rely on comparing memory fingerprints to estimate this information. Since a Bloom filter is an approximate data structure that can yield false positives, we can estimate the exact the exact sharing rate between two or more virtual machines. However, they can still provide sufficient accuracy for our purpose and at a significantly reduced computational cost than comparing hash lists.

The typical use for a Bloom Filter is to efficiently test if a single key is contained within a data set. In our work, we use Bloom Filters for a different purpose—to efficiently determine the total number of keys that are present in both data sets. The simplest method to estimate the number of common elements in two Bloom filters is to calculate the inner product of their bit vectors. Intuitively, if a key has been inserted into both Bloom filters, then the same bits will be set to 1 in both cases since Bloom filters always a set and never clear bits. When the bitwise AND operation is applied to the two filters, those bits will remain set. Thus the number of set bits is an indicator of the number of common keys present in both—the larger the number of set bits in the resulting AND, the greater the number of shared keys. This approach has been successfully used in information retrieval research for matching similar document lists [10] . Although the AND operation yields an indication of the extent of sharing between the two Bloom Filters, our system requires a quantitative estimate of the number of shared elements. To derive such an estimate, we use a well-known result from Bloom filter theory that estimates the number of shared elements using the following equation [15]:

$$share = \frac{ln(z_1 + z_2 - z_{12} - ln(z_1 * z_2) + ln(m)}{k(ln(m) - ln(m-1))} \quad (2)$$

where z_1 and z_2 are the numbers of zeros in the two Bloom filters, z_{12} is the number of zeros in the AND of the two filters, m is the size of the filters, and k is the number of hash functions used. *share* yields the number of keys common to the two Bloom Filters.

Our Memory Buddies system uses the above result to compare a VM fingerprint to the composite fingerprint of a physical server to estimate how many pages of the

VM are already present on the server and thus can be shared if the VM were to be moved to that server.

Observe that estimating sharing using Equation 2 is far more computationally efficient than comparing hash lists. Whereas the latter is an $O(n \cdot logn)$ operation, the former involves bit vector manipulation, which can be performed very efficiently. This overhead is important since a large data center may contain hundreds or thousands of physical servers and virtual machines, and our system needs to compare a VM's fingerprint with that of each physical server in the data center to identify the best sharing potential—which can involve several thousand fingerprint comparisons per VM.

4.2 Consolidation Algorithm

Our server consolidation algorithm opportunistically identifies servers that are candidates for shutting down and attempts to migrate virtual machines off of them to hosts with high sharing opportunities. In doing so, it attempts to pack VMs onto servers so as to reduce aggregate memory footprint and maximize the number of VMs that can be housed in the data center. Once the migrations are complete, the consolidation candidates can be powered down until new server capacity is needed, thereby saving on operational (energy) costs. The consolidation algorithm comprises three phases:

Phase 1: *Identify servers to consolidate.* The consolidation algorithm runs periodically (e.g., once a day) and can also be invoked manually when needed. Our system assumes that the various nuclei periodically report memory fingerprints as well as the resource usages on each server. Monitored resources include memory, CPU, network bandwidth and disk; both the mean usage over the measurement interval as well as the peak observed usage are reported. The algorithm uses these reported usages to identify servers to consolidate; a server becomes a candidate for consolidation if its mean usage remains below a low threshold τ_{low} for an extended duration.[3] Currently our system only considers memory usages when identifying consolidation candidates; however, it is easy to extend it to check usages of all resources to identify lightly loaded servers.

Phase 2: *Determine target hosts.* Once the set of consolidation candidates have been identified, the algorithm must determine a new physical server to house each VM. To do so, we order VMs in decreasing order of their memory sizes and consider them for migration one at a time. For each VM, the algorithm first determines the set of feasible servers in the data center. A feasible server is one that has sufficient available resources to house that VM—recall that each VM is allocated a slice of the CPU, network bandwidth and memory on the server, and only servers with at least this much spare capacity should be considered as possible targets. Given a set of feasible servers, the algorithm must estimate the page sharing potential on each host using our fingerprint comparison technique—the fingerprint for the VM is compared with the composite fingerprint of the physical server to estimate the number of shared pages using Equation 2. The algorithm then simply chooses the server that offers the maximum sharing potential as the new host for that VM.

[3]In addition, the system can also check that the peak usage over this duration stayed below a threshold, to ensure that the server did not experience any load spikes during this period.

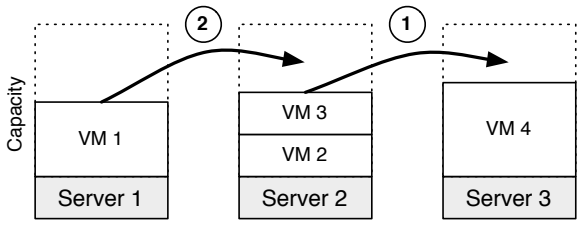


Figure 3: A 2-step migration.

In certain cases, it is possible that there are no feasible servers for a VM. This can happen if the VM has a large CPU, network or memory footprint and no existing servers has sufficient idle capacity to house it, or if existing servers in the data centers are heavily utilized. In either case, the consolidation algorithm is unable to directly migrate the VM to another host. Instead it must consider a a multi-way move, where one of more VMs from an existing server are moved to other servers to free up additional capacity and make this server feasible for the VM under consideration. Currently, we pick the server with the maximum spare capacity and determine how much additional capacity needs to be freed up to make that server a feasible host for the VM. One or more existing VMs are then chosen for migration so that at least this much capacity is freed up. A feasible server that offers the highest sharing potential is chosen as the new host for each such VM.

Figure 3 illustrates such a multi-way move where VM_1 is considered for consolidation but no server has sufficient spare memory to house this VM. Since server 2 has the most spare capacity, the algorithm determines that it needs to free up additional 128MB of memory and decides to move VM_3 to server 3. The server then becomes feasible for VM_1.

In certain rare cases, even multi-way moves may not be feasible—for instance, when the data center servers are heavily loaded. In such a scenario, rather than considering more complex k-way swaps, our algorithms indicates a failure to free up the consolidation candidate and moves on to the next consolidation candidate.

Phase 3: *Migrate VMs to targets.* Once new destinations have been determined for each VM on the consoli-

dation servers, our algorithm can perform the actual migrations. Live migration is used to ensure transparency and near-zero down-times for the application executing inside the migrated VMs.

To ensure minimum impact of network copying triggered by each migration on application performance, our algorithm places a limit on the number of concurrent migrations to no more than C at a time; once each migration completes, a pending one is triggered until all VMs have migrated to their new hosts. The original servers are then powered off.

5. HOTSPOT MITIGATION

The hotspot mitigation technique works in conjunction with the consolidation mechanism to prevent instability caused by packing virtual machines too tightly. Its primary goal is to eliminate memory hotspots resulting from excessive swapping. Although other kinds of hotspots can occur due to CPU, network and disk overloads, this works focuses on memory hotspots and techniques such as those in [26] can be employed to deal with other kinds of hotspots. We define a memory hotspot as swapping activity that exceeds a threshold and which results from increased memory pressure. Our system must detect such hotspots when they form and mitigate their effects by rebalancing the load in the data center.

Typically an increase in memory pressure can result from two types of effects. First, the aggregate memory needs of application can increase (e.g., if it `mallocs` a large amount of memory); if the memory slice allocated to the VM is insufficient to meet these increased needs, then guest OS must resort to swapping pages from disk. In this case, the VM needs to be allocated a larger slice of memory on the underlying physical server to reduce memory pressure. The second effect is caused when an application that is sharing a large number of pages with another VM sees a major phase change. For instance, VM may terminate the application, which causes shared pages to be deallocated, or the application may change it execution behavior, which causes shared pages to be swapped out or become unshared due to copy-on-writes. In either case, the amount of sharing on the physical server decreases, which in turn increases the aggregate memory footprint of the VMs. Since our consolidation algorithm often overbooks memory,[4] the resulting increase in memory pressure can cause swapping. We must distinguish between these two types of memory hotspots, since a different set of actions are needed to mitigate their effects. We refer to the two types of hotspots as *type 1* and *type 2* hotspots, respectively.

The control plane relies on the statistics reported by the nuclei to detect both type 1 and type 2 hotspots. In particular, each nucleus is assumed to monitor the swap activity of each guest OS as well as the number of shared pages on the physical server; these statistics are included in the periodic reports to the control plane, along with the memory fingerprints. If the reported swap activity of a virtual machine rises above a threshold but there is no significant change in the number of shared pages, a type 1 hotspot is flagged by the control plane. In contrast, if the number of shared pages declines when compared to previously reported values and swapping is observed (or imminent), then a type 2 hotspot is flagged.

To handle a type 1 hotspot, the control plane must estimate how much additional memory should be allocated to the virtual machine to eliminate swapping. Type 1 hotspots occur because the virtual machine requires more memory than it is guaranteed by its SLA based slice. Other load balancing systems such as [26, 21] exist which deal with this situation by increasing memory allocations when swapping is detected. We focus on type 2 hotspots which are directly related to page sharing.

To handle a type 2 hotspot, the control plane attempts to migrate the VM that triggered loss of page sharing to another host. In this case, a feasible server is picked at random and the VM is migrated to that host. Page sharing potential is *not* considered when picking a feasible server since the VM has seen a recent phase change in its memory behavior, and no long-terms statistics are available to make an informed decisions about the VM's future page sharing potential.

If there are no feasible destinations for the virtual machines on the overloaded host, a server must be brought in from the shutdown pool so that it can host one or more of the VMs.

6. IMPLEMENTATION

The Memory Buddies implementation uses VMware ESX server for the virtualization layer as it supports both page sharing and virtual machine migration. We have implemented our memory tracing software as a modification of the Linux 2.6.20 kernel. The kernel modifications span seven files and include approximately 400 lines of code. The kernel daemon gathers memory utilization statistics and page hashes which are exported to a user space application. The data is transmitted to the control plane each measurement interval, currently every 20 seconds.

The control plane is a Java based server which communicates with the VMware Virtual Infrastructure management console via a web services based API. The API is by the control plane to discover which hosts are cur-

[4]By overbooking, we mean that the total footprint of the VMs exceeds physical RAM in the absence of sharing but sharing allows us to fit these VM in RAM. e.g., if there are two VMs, $M_1 + M_2 > RAM$ but with S shared pages we get $M_1 + M_2 - S \leq RAM$.

rently active and where each virtual machine resides. Extra resource statistics are retrieved from the VMware management node such as the total memory allocation for each VM. This API is also used to initiate virtual machine migrations between hosts. The control plane primarily consists of statistic gathering, sharing estimation, and migration components which comprise about 3600 lines of code.

Our use of a Linux based memory tracer currently prevents us from testing Memory Buddies on Windows virtual machines. However, all of the components of the memory tracer could be included within the hypervisor layer to make it platform independent. We have been unable to implement our software within the virtualization layer due to the closed source nature of VMware's ESX server. The key components of the memory tracer are the page hash generation function and access bit sampling component used to generate the LRU list. Page hashing is already performed by the hypervisor to detect sharing, however the data is not exported to other applications, requiring us to gather our own hashes [24]. The memory tracing algorithm we use has been implemented within the ESX Server virtualization layer by one of the authors at VMware, however this is not currently available as a commercial product, so we were unable to use it in our system [4].

7. EXPERIMENTAL EVALUATION

We have evaluated the Memory Buddies system to study the gains in consolidation which are possible when utilizing page sharing to guide virtual machine placement. Our experiments have been performed on a cluster of four P4 2.4Ghz servers connected over gigabit ethernet. Each server ran ESX Server 3.0.1 and the VMware Virtual Infrastructure 2.0.1 management system ran on an additional node.

Our experiments utilize a range of realistic data center applications:

- RUBiS is an open source multi-tier web application that implements an eBay-like auction web site and includes a workload generator that emulates users browsing and bidding on items. We use the Apache/PHP implementation of RUBiS version 1.4.3 with a MySQL database.
- TPC-W models an Amazon style e-commerce website implemented with Java servlets and run on the Jigsaw server with a DB2 backend.
- SpecJBB 2005 is a Java based business application benchmark which emulates a 3-tier system with a workload generator.
- Apache Open For Business (OFBiz) is an open source suite of enterprise web applications with accouting, finance, and sales functionality used by many businesses. We utilize the eCommerce component and a workload generator based on the JWebUnit testing framework to emulate client browsing activities.

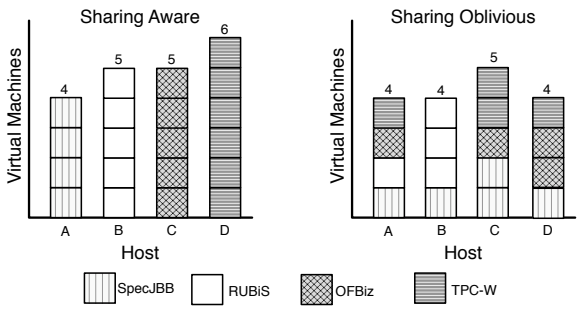


Figure 4: Sharing aware placement groups similar virtual machines.

Application	Approx. Sharing
TPC-W	38%
OpenForBiz	18%
RUBiS	16%
SpecJBB	9%

Table 1: Application types and sharing levels

For the multi-tier applications, we run all tiers within a single virtual machine. All Apache web servers are version 2.2.3 with PHP 4.4.4-9, MySQL databases are 5.0.41, Jigsaw is version 2.2.6, and the DB2 server was DB2 Express-C 9.1.2.

In our experiments we compare two placement algorithms. Our *sharing aware* approach attempts to place each virtual machine on the host that will maximize its page sharing. The *sharing oblivious* scheme does not consider sharing opportunities when placing virtual machines, and instead places each virtual machine on the first host it finds with sufficient spare capacity. Although the sharing oblivious approach does not explicitly utilize sharing information to guide placement, page sharing will still occur if it happens to place virtual machines together with commmon pages.

7.1 Server Consolidation

7.1.1 Sharing Based Placement

We first test Memory Buddies' ability to use sharing information to more effectively place servers on a testbed of four hosts. We utilize four different applications to vary the sharing rate between virtual machines. Table 1 lists the different application types and their approximate level of sharing on a 384MB virtual machine; actual sharing values vary within a few percent depending on paging activities. Each virtual machine runs one of these applications, and while the core ap-

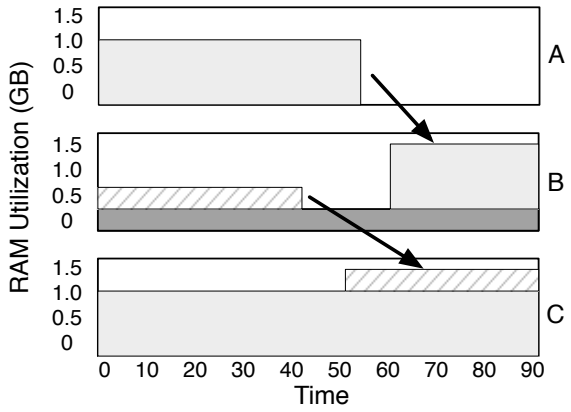


Figure 5: Two Way Migration.

plication data is identical, the workloads and database contents are unique for each VM.

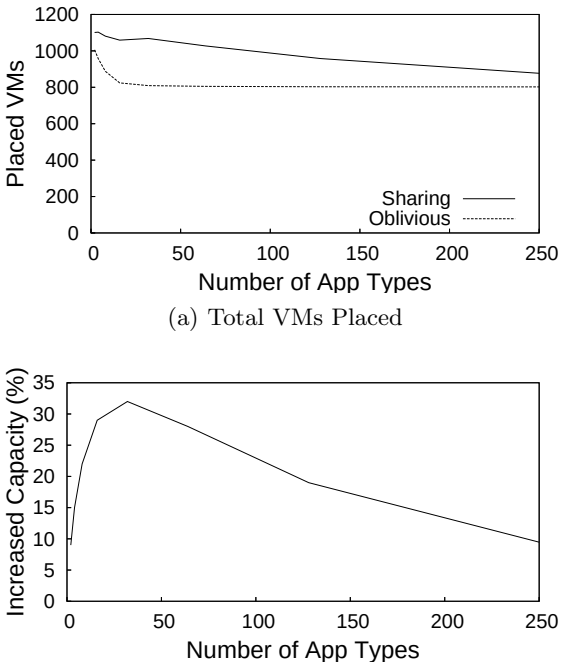
Initially, we create one virtual machine of each type and place it on its own physical host. Additional VMs of each type are then spawned on a fifth host and migrated to one of the four primary hosts. We compare the number of virtual machines which can be successfully hosted using both our sharing aware algorithm which migrates each new VM to the host with the greatest sharing potential and a sharing oblivious placement algorithm which migrates each VM to the first host it finds with sufficient memory, without regard to sharing. The experiment terminates when no new virtual machines can be placed.

Each virtual machine is configured with 384 MB of RAM, and the hosts have 1.5 GB of spare memory since VMware reserves 0.5 GB for itself. Thus we expect each host to be able to run about four VMs without sharing. Figure 4 displays the final placements reached by each algorithm. The three web applications, TPC-W, OFBiz, and RUBiS, demonstrate a benefit from utilizing sharing, allowing more VMs to be packed than the base four. The sharing oblivious algorithm places four VMs on each host, except for host C on which it fits an extra VM due to the sharing between TPC-W instances. The sharing aware approach is able to place a total of 20 virtual machines, while the Oblivious approach can only fit 17. For this scenario, exploiting sharing increased the data center's capacity by a modest 17%.

Result: Memory Buddies can detect which applications have similar memory contents and automatically place them together. By reducing the total memory requirements on each host, the effective capacity of each host can be increased.

7.1.2 Consolidation through Multi-Way Moves

This experiment demonstrates how Sandpiper can initiate multi-way migrations in order to increase the chance



(a) Total VMs Placed

(b) Capacity Gain over Oblivious

Figure 6: Benefits of exploiting sharing in a 100 node data center.

that a server can be fully consolidated. We run a web hosting type scenario where each virtual machine on three hosts runs Apache serving a set of static web pages. We imagine a tiered payment system where "Gold" class virtual machines are allocated 1024MB of RAM, "Silver" class machines receive 384MB and each host has a total of 1.5GB of memory available. In the initial setup, hosts A and B each run a single Gold class virtual machine, while host C runs two Silver virtual machines. Host A is targetted for consolidation so that the server can be shut down for maintenance. Unfortunately, neither host B nor C have sufficient spare capacity to accomodate the Gold VM from host A. Memory Buddies detects this and determines that since host B has the most spare capacity, space should be made on it to allow the consolidation to proceed. Figure 5 shows that, as a result, one of the Bronze VMs is migrated from host B to host C at time 40 seconds. This migration takes approximately 10 seconds to complete and allows the Gold VM from A to be safely migrated to B.

Result: Memory Buddies can detect when an attempt at consolidation fails due to lack of space and can try to resolve this by using a two-step migration. This increases the chance that all VMs can be safely migrated from a host, allowing more hosts to be fully shut down.

7.1.3 Data Center Consolidation

This experiment simulates a large data center of 100

hosts to demonstrate how the sharing aware placement algorithm can lead to substantial consolidation benefits.

We consider a data center with a mixed workload of Windows and Linux based virtual machines. The virtual machines run one of a variable number of applications which exhibit either high, medium, or low sharing. The majority of each virtual machine's memory is VM-specific data which is unlikely to be sharable. The result is a set of virtual machines which can share a small amount of memory with other VMs running the same operating system, and a larger amount of memory with VMs also running the same application.

To simulate this kind of realistic environment, an artificial memory fingerprint is created for each virtual machine by generating a series of hashes representing the VM's OS, application, and data pages. First a set of pages are created for the OS which provide a baseline of 5% sharing between virtual machines of the same operating system. The next portion of the fingerprint is filled with page hashes representing the VM's application. We vary the number of possible applications to study the impact of application diversity. Of these applications, one third have 40% sharing, one third have 25% sharing and the rest have 10% sharing. Unmatched applications have negligible sharing. The remainder of the fingerprint is filled with unique hashes to represent the VM's data pages. Each host is created with 4GB of memory and each virtual machine uses 512MB, so we expect a base line of eight virtual machines per server when there is very little sharing.

When there are a small number of application types, explicitly considering sharing potential does not provide a large gain since a Oblivious algorithm still has a high chance of placing similar applications together. As a result, Figure 6(a) shows that the total number of virtual machines which can be placed in the data center with either the sharing aware or sharing oblivious algorithm is similar when there is a very small number of application types.

As the number of application types begins to rise, the relative benefit of using sharing increases as the chance of randomly achieving a good placement falls. This is demonstrated in 6(a) by how rapidly the sharing oblivious approach drops back to only placing the baseline of 8 virtual machines per server. In contrast, the sharing aware approach continues to successfully place a large number of virtual machines on each server. The peak benefit occurs around 30 application types, where the Sharing Aware algorithm can place 32% more virtual machines than sharing oblivious. In our 100 node data center, this results in an additional 147 virtual machines being hosted when sharing guides placement.

When the number of application types becomes very high relative to the number of hosts in the data center, utilizing sharing becomes less effective since more application types must be mixed on each server. Figure 6(b) shows how the increased capacity achieved by exploiting sharing sharply rises and then gradually falls as the number of application types increases.

Result: The potential gain from using sharing ranges from 10 to 30 percent depending on application diversity. Using sharing to guide placement accrues the most significant benefts when there is a low or medium ratio of application types to hosts.

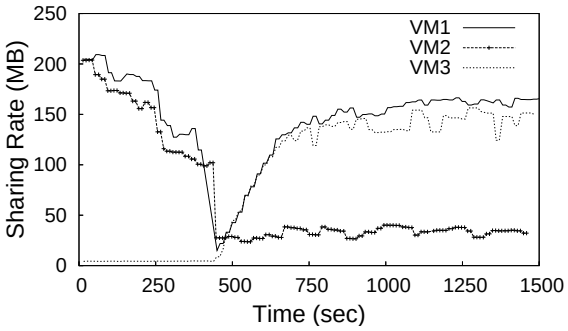
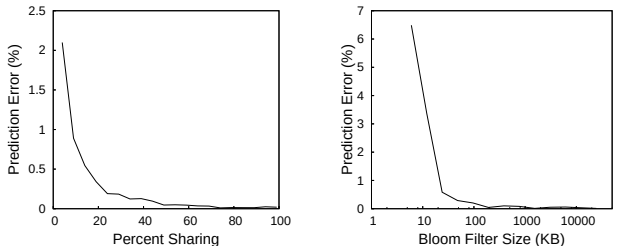


Figure 7: Hotspot mitigation.

7.2 Hotspot Mitigation

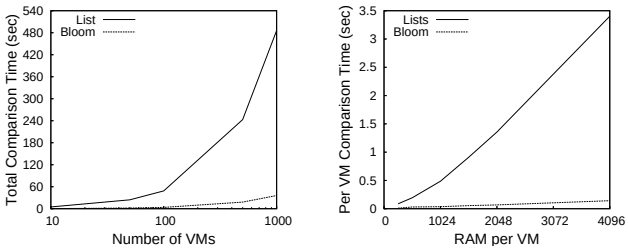
This experiment demonstrates Memory Buddies' ability to detect and respond to a type 2 memory hotspot– when application phase changes reduce the potential for sharing, requiring action to be taken to prevent hurting application performance. The experiment employs two hosts, the first running two virtual machines and the second running only one. All of the virtual machines run Apache web servers, and initially an identical set of requests are sent to each, resulting in a high potential for sharing. Figure 7 shows the number of pages shared by each server over time. Since VM_1 and VM_2 reside on the same host, they initially have a high level of sharing. At time 60 seconds, a phase change occurs for the requests being sent to VM_2. As a result, the sharing between VM_1 and VM_2 decreases significantly. However, since VM_1 and VM_3 continue to receive the same workload, there is a high potential for sharing between them. The Memory Buddies system detects that this is a type 2 hotspot since the sharing potential has decreased but there is no detected swapping or change in the VM's LRU list. As a result, the system determines that VM_1 should be migrated to Host 2 at time 360 seconds. After the migration completes, the sharing rate between VM_1 and VM_3 gradually increases again.

Result: Memory Buddies' monitoring system is able to detect changes in sharing potential brought on by application phase transitions. This type of hotspot is automatically resolved by determining a different host with a higher sharing potential for one of the VMs.



(a) Sharing Rate (b) Bloom Filter Size

Figure 8: Bloom filter accuracy.



(a) Comparison Method (b) VM Memory

Figure 9: Fingerprint comparison efficiency.

7.3 Bloom Filter Accuracy

This experiment demonstrates the accuracy of performing fingerprint comparisons using Bloom Filters. We first generate two sets of page hashes with a configruable amount of sharing using a random number generator. The size of the list is equivalent to a virtual machine with 512MB of RAM, and the hashes are inserted into a Bloom Filter with a 384KB bit vector. The intersection of the two lists is estimated using our Bloom Filter approach and then validated using an exact list comparison. Figure 8(a) shows how the bloom filter's accuracy varies depending on the level of sharing between the two lists. With a lower level of sharing, hash collisions are more likely to reduce the accuracy of the Bloom Filter comparison method by suggesting that non-existant pages will be sharable. As sharing increases, the two bloom filters will have greater similarity, resulting in fewer false positives impacting the sharing estimate. Since our system is primarily interested in finding the maximum sharing point, reduced accuracy at lower sharing levels will have little effect on the choices made by the system.

Next we measure the accuracy of Bloom Filter comparisons when the size of the Bloom Filter's bit vector is varied. Using two sets of page hashes gathered from TPC-W applications, we test Bloom Filters with sizes ranging from 6KB to a maximum size of 24MB. The hash lists come from two virtual machines with 384MB of RAM that exhibited approximately 38% sharing. Figure 8(b) illustrates how the comparison accuracy rapidly decreases as filter size rises, and that there is little benefit from utilizing Bloom Filters beyond a few hundred KB.

Result: Bloom filters can provide a high level of accuracy for estimating the number of identical elements in two sets, particularly when the actual set intersection is high.

7.4 Fingerprint Comparison Efficiency

Next we compare the computation cost of using Bloom filters to using hash lists. Figure 9(a) plots the time to calculate the sharing potential between one VM and and all other hosts in data centers of varying size. In this simulation, each host in the data center has two VMs (each with 1024MB ram). An additional host runs only a single virtual server, VM_1, and is targetted as a candidate for consolidation. The consolidation algorithm is run and compares the sharing potential between VM_1 and each VM on the other hosts in the data center. We measure the time to perform these comparisons using both our Bloom Filter based approach and with a direct comparison of hash lists. Our results show that Bloom filters are faster by more than an order of magnitude compared to using a brute force comparison of hash lists. Within one minute, the Bloom Filter based approach can compare as many as 1,700 virtual machines, while the hash list approach can only compare 125. Much of the cost in list comparison is due to sorting time required to order the lists. The comparison time with Bloom Filters is independent of the list order.

The sharing estimation time is based not only on the number of virtual machines in the data center, but also on the amount of memory dedicated to each virtual machine. When using hash lists, a virtual machine with a larger memory image will have a correspondingly larger list of hashes which need to be sorted and compared. For a given Bloom Filter, the computation time is not directly related to the number of memory pages, but as the amount of memory increases, larger Bloom Filters are required in order to maintain a target level of accuracy. Figure 9(b) demonstrates how the comparison time for a single VM increases with memory size. Here the Bloom Filter is resized to maintain a constant error rate as the memory size increases. This indicates that hash list comparisons are simply infeasible when using virtual machines with large memory sizes. For example, using hash lists took 3.4 seconds *for each virtual machine comparison* when using 4GB hosts, compared to 0.142 seconds using Bloom Filters.

Result: The Bloom Filter based fingerprint comparison is more than an order of magnitude faster than direct hash list comparisons, particularly when compar-

VM RAM	Hash List Size (KB)	Bloom Size (KB) w/Error 1%	0.1%	0.01%
1GB	1024	306	460	613
4GB	4096	1227	1840	2454
8GB	8192	2455	3680	4908

Table 2: Per VM communication cost in KB for hash lists and Bloom Filters with varying accuracy.

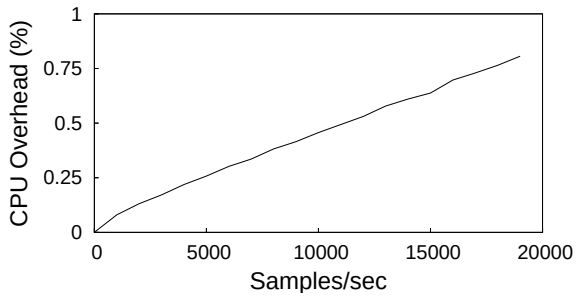


Figure 10: Memory Tracer CPU overhead.

ing virtual machines with large memory sizes.

7.5 System Overheads

Here we examine Memory Buddies scalability by looking at its CPU and network overheads. The total communication overhead of the system is dependent on the number of VMs running in the data center. Table 2 compares the cost of storing or transmitting Bloom Filter based memory fingerprints or hash lists of various sizes. The Bloom Filter sizes are based on maintaining a 0.01% false positive rate. Although our experiments utilize an aggressive 20 second monitoring interval, a much coarser grain monitoring window would be sufficient in a real data center due to the slow rate at which sharing typically adjusts. It can take VMware tens of minutes to fully identify the sharable pages between two virtual machines, so more frequent updates are only useful for detecting potential changes in sharing causing hot spots. To further reduce communication costs, the Nucleus could estimate the difference between two successively generated Bloom Filters to determine if there has been sufficient change to warrant forwarding the information to the control plane.

The computation performed on the control plane is dominated by the cost of fingerprint comparison, but as demonstrated in Figure 9(a), this scales well to large data centers when using Bloom Filters. The memory tracer incurs some overhead on each virtual machine since CPU time must be sent checking the access bits of pages and maintaing the LRU list. The CPU overhead of the memory tracer is controlled by the sampling rate as shown in Figure 10. We utilize a rate of 10,000 samples per second which incurs less than 0.5% overhead.

8. RELATED WORK

A content based page sharing scheme was first proposed in the Disco [2] system, but required modification of operating system calls for page creation and modification. When and was later implemented in both the VMware and Xen virtualization platforms [24, 12]. Both approaches use a background hashing mechanism in the virtualization layer to detect pages in guest VMs with identical contents. Since we lack access to the hashes gathered by the VMware hypervisor, we mimic this functionality within the guest OS. The Memory Buddies system additionally provides a mean to detect the potential for sharing between virtual machines on separate hosts.

Several memory monitoring schemes have been proposed for use in virtual environments. The Geiger system [11] presented a method for monitoring a virtual machine's unified buffer cache to infer information about the virtual memory system and working set. Geiger combines monitoring of swap disk activity with page eviction events to determine how the guest is utilizing its buffer cache. Lu and Shen propose a hypervisor based cache which uses a similar technique of monitoring how guests are swapping out pages to make estimates of working set size. The overhead of the system is reduced since pages are swapped from the guest to an in-memory hypervisor cache rather than to disk. This information is used for an adaptive memory allocation scheme which grants memory to virtual machines to balance swap rates. Memory Buddies estimates an approximate LRU curve using an access bit based memory tracer proposed in [3]. A similar memory tracer has been implemented within the VMware hypervisor [4].

Process migration was first investigated in the 80's [18, 23]. The re-emergence of virtualization led to techniques for virtual machine migration performed over long time scales in [20, 25, 13]. The means for "live" migration of virtual machines incurring downtimes of only tens of milliseconds have been implemented in both Xen [5] and VMWare [17]. At the time of writing, however, only VMWare's ESX server supports both live migration and page sharing simultaneously.

Virtual machine migration was used for dynamic resource allocation over large time scales in [19, 22, 7]. Previous work [26] and the VMware Distributed Resource [21] monitor CPU, network, and memory utilization in clusters of virtual machines and use migration for load balancing. The Memory Buddies system is designed to work in conjunction with these sorts of multi-resource load balancing systems by providing a means to use page sharing to help guide placement decisions.

Bloom filters were first proposed in [1] to provide a tradeoff between space and accuracy when storing hash coded information. Guo et al. provide a good overview of Bloom Filters as well as an introduction to intersection techniques [8]. Bloom filters have also been used to rapdily compare search document sets in [10] by comparing the inner product of pairs of Bloom Filters. The Bloom Filter intersection technique we use provides a more accurate estimate based on the Bloom Filter properties related to the probability of individual bits being set in the bit vector. This approach was used in [15] to detect similar workloads in peer to peer networks.

9. CONCLUSIONS

Modern data centers are increasingly employ a virtualized architecture in order to maximize resource utilization while reducing costs. With the advent of multicore processors, memory has become the key limiting resource when running multiple virtual machines on a given server. In this paper, we argued that server consolidation techniques are often limited because memory cannot be allocated as flexibly as CPU or network resources and advocated the use of page sharing techniques to significantly reduce aggregate memory requirements of collocated VMs and increase the number of VMs that can be housed on a given data center. We presented a memory tracer and a novel memory fingerprinting technique—based on Bloom Filters— to concisely and efficiently capture the memory contents of each VM. Our server consolidation algorithm employs a fast, scalable fingerprint comparison technique to efficiently determine which data center servers offer the maximum sharing potential for a given VM, and consolidates these VMs onto these servers via live migration. We also presented a hotspot mitigation technique to correct the negative impacts of unanticipated loss of page sharing and swapping. We implemented a prototype of our Memory Buddies system in the VMware ESX server. Using a mix of enterprise and ecommerce applications, we showed that our Memory Buddies system is able to increase the effective capacity of a data center by over 30% when consolidating VMs from underloaded servers. We also showed that our system can effectively detect and resolve memory hotspots due to changes in sharing patterns. Our system imposes minimal CPU and network overheads of less than 0.5% per server and can scale well to large data centers with hundreds of machines. As part of future work, we plan to enhance our techniques to dynamically vary memory allocations, possibly by migrating virtual machines, to improve memory utilization and further increase the number of VMs that can be placed on a data center while guaranteeing application performance.

10. REFERENCES

[1] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[2] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *SOSP*, pages 143–156, 1997.

[3] James Cipar, Mark D. Corner, and Emery D. Berger. Transparent Contribution of Memory (Short Paper). In *Proceedings of USENIX Annual Technical Conference*, pages 109–114, Boston, MA, May 2006.

[4] Jim Cipar and Yuri Baskakov. Miss ratio curve estimation by accessed-bit sampling. In *VMWorld 2007 Academic Poster Session*.

[5] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfiel. Live migration of virtual machines. In *Proceedings of Usenix Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition edition, 2001.

[7] Laura Grit, David Irwin, , Aydan Yumerefendi, and Jeff Chase. Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In *In the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November 2006.

[8] Deke Guo, Jie Wu, Honghui Chen, and Xueshan Luo. Theory and network applications of dynamic bloom filters. In *INFOCOM*, 2006.

[9] Paul Hsieh. Hash functions.

[10] Navendu Jain, Michael Dahlin, and Renu Tewari. Using bloom filters to refine web search results. In *WebDB*, pages 25–30, 2005.

[11] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 14–24, New York, NY, USA, 2006. ACM Press.

[12] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. On the feasibility of memory sharing: Content-based page sharing in the xen virtual machine monitor, June 2006.

[13] M. Kozuch and M. Satyanarayanan. Internet suspend and resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY*, June 2002.

[14] Purushottam Kulkarni, Prashant J. Shenoy, and Weibo Gong. Scalable techniques for memory-efficient cdn simulations. In *WWW*, pages 609–618, 2003.

[15] Xucheng Luo, Zhiguang Qin, Ji Geng, and Jiaqing Luo. Iac: Interest-aware caching for unstructured p2p. In *SKG*, page 58, 2006.

[16] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[17] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX Annual Technical Conference*, 2005.

[18] M. Powell and B. Miller. Process migration in DEMOS/MP. *Operating Systems Review*, 17(5):110–119, 1983.

[19] Paul Ruth, Junghwan Rhee, Dongyan Xu, Rick Kennell, and Sebastien Goasguen. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *IEEE International Conference on Autonomic Computing (ICAC)*, June 2006.

[20] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[21] VMware Distributed Resource Scheduler.

[22] A. SUNDARARAJ, A. GUPTA, and P. DINDA. Increasing Application Performance in Virtual Environments through Run-time Inference and Adaptation. In *Fourteenth International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[23] M. M. Theimer, K. A. L., and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-System. pages 2–12, December 1985.

[24] C. Waldspurger. Memory Resource Management in VMWare ESX Server. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)*, December 2002.

[25] A. Wohitaker, M. Shaw, and S. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[26] Timothy Wood, Prashant J. Shenoy, Arun Venkataramani, and Mazin S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.