A Policy-Based System for Dynamic Scaling of Virtual Machine Memory Reservations

Rebecca Smith
Rice University
rjs@rice.edu

Scott Rixner
Rice University
rixner@rice.edu

ABSTRACT

To maximize the effectiveness of modern virtualization systems, resources must be allocated fairly and efficiently amongst virtual machines (VMs). However, current policies for allocating memory are relatively static. As a result, system-wide memory utilization is often sub-optimal, leading to unnecessary paging and performance degradation. To better utilize the large-scale memory resources of modern machines, the virtualization system must allow virtual machines to expand beyond their initial memory reservations, while still fairly supporting concurrent virtual machines. This paper presents a system for dynamically allocating memory amongst virtual machines at runtime, as well as an evaluation of six allocation policies implemented within the system. The system allows guest VMs to expand and contract according to their changing demands by uniquely improving and integrating mechanisms such as memory ballooning, memory hotplug, and hypervisor paging. Furthermore, the system provides fairness by guaranteeing each guest a minimum reservation, charging for rentals beyond this minimum, and enforcing timely reclamation of memory.

CCS CONCEPTS

• **Software and its engineering → Virtual machines**; **Operating systems**; **Memory management**;

KEYWORDS

Virtualization; Memory management; Dynamic memory allocation

ACM Reference Format:
Rebecca Smith and Scott Rixner. 2017. A Policy-Based System for Dynamic Scaling of Virtual Machine Memory Reservations. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3127491

1 INTRODUCTION

Server consolidation via virtualization is a mainstay of cloud computing technologies. Virtualization enables cloud providers to more efficiently allocate and utilize physical computing resources by decoupling the virtual machines' resource demands from the physical machine's resources. Virtual machine performance and cloud provider cost efficiency both depend heavily on the resource allocation policies of the virtualization system. As workloads are rarely static, it is important for virtualization solutions to be able to adjust resource allocation accordingly. Most efforts in scaling resource allocation, however, focus on dynamically changing the number of guest VMs (for example, [3, 5, 6, 24]) rather than scaling the resources of an individual VM.

Memory is one of the critical resources in modern virtualization systems. This paper focuses on the dynamic assignment of memory to guest VMs running on a single physical machine. Currently, in common virtualization systems such as Xen and KVM, an administrator may manually re-balance the memory allocation amongst guests. However, neither Xen nor KVM currently ships with any tools to enable fair and automatic reallocation of memory to guest VMs in response to dynamically changing demands. This lack of flexibility can lead to paging by the guest operating systems, resulting in performance degradations that could have been prevented by reallocating memory at runtime.

Past work has looked at extending virtualization systems to dynamically reallocate memory amongst guest VMs [2, 11, 15, 22, 23, 25, 27]. However, these proposals suffer from several limitations, and none have been adopted within mainline Xen or KVM. First, the mechanisms and policies within these proposals are tightly coupled. If the user wishes to take advantage of the mechanisms, they have no freedom to utilize different policies that might be more appropriate in their particular deployment. Furthermore, the policies cannot be easily evolved, as they are intertwined with the mechanisms upon which they were implemented. Second, all of these proposals assume a static maximum memory allocation for virtual machines that is fixed at boot time. Third, these proposals are targeted at small workloads, on the order of 1 GB. Finally, some of these proposals are directly tied to the characteristics of specific workloads.

To enable runtime reallocation of large amounts of memory while addressing the limitations of previous efforts, this paper introduces a policy-based system for dynamically scaling VM memory reservations based on their changing demands. While the system has been implemented within Xen, its mechanisms and policies transcend any particular virtualization system. The key component of this system is the new `flexmemd` daemon that runs in Xen's driver domain (Domain 0) and responds to changes in guest demands by determining if and how memory should be reallocated. While `flexmemd` guarantees a minimum memory allocation for each guest, it does not impose a maximum. A guest can therefore expand beyond its initial allocations if (1) it is beneficial for it to do so, (2) there is sufficient memory capacity in the system to allow it, (3) it is fair to do so, and (4) `flexmemd` deems it best to allocate extra memory to this particular guest.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '17, September 24–27, 2017, Santa Clara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3127491

Once `flexmemd` computes a desired partitioning of memory amongst the guests, it enforces this partitioning by uniquely coordinating a set of mechanisms including memory ballooning, memory hotplug, and hypervisor paging. If a guest is granted memory beyond its initial physical memory allocation, it utilizes memory hotplug to acquire additional resources. If a guest is required to release memory back to the system but its balloon driver does not comply quickly enough, hypervisor paging forcibly reclaims memory from that guest. Guests are therefore incentivized to release memory when asked to do so.

The contributions of this paper are as follows:

- **The integration and improvement of mechanisms across the virtual machine monitor and guest OS.** Current ballooning mechanisms are slow to react and are only accessible via manual administration. The existing mechanisms have been upgraded and integrated in novel ways in order to automatically reallocate memory quickly enough to be useful for modern, large-scale workloads.

- **The design and implementation of `flexmemd`.** The `flexmemd` daemon integrates the aforementioned mechanisms to implement fair and efficient memory reallocation at runtime. It has a decoupled architecture that separates mechanisms from policies to enable the creation of a diverse set of memory reallocation policies. This daemon is independent of the guest OS and workload.

- **Effective memory reallocation policies for large, modern workloads.** The `flexmemd` daemon is flexible enough to support both simple and complex policies, such as novel policies that utilize direct assignment and sealed bid auctions.

- **Implementation and evaluation of the system across multiple guest operating systems.** `flexmemd` supports a simple interface to guest operating systems, enabling easy interoperability. Unlike past work, multiple operating systems, Linux and OSv, were modified to work with `flexmemd`, demonstrating this interoperability.

The rest of this paper proceeds as follows. Section 2 presents background on existing mechanisms that can be used to change guests' memory allocations. Sections 3 and 4 present the new mechanisms and policies, respectively, used by the system to dynamically assign memory. Section 5 evaluates the system, Section 6 presents related work, and Section 7 concludes the paper.

2 BACKGROUND

The system has been implemented within Xen, and currently works with both Linux and OSv guests. However, the concepts are applicable to any operating system. The system makes use of modified versions of several existing mechanisms within these systems, including the Linux balloon driver and several utilities that run within Xen's driver domain.

2.1 Memory Ballooning

Ballooning allows the hypervisor to dynamically add or remove memory from running guests, with the cooperation of the guests themselves [22]. Memory is reclaimed by inflating the balloon. This is a process whereby the balloon driver in the guest allocates pinned memory and gives it to the hypervisor, which then unmaps it from the guest. The hypervisor can then reallocate that memory to other guests. To restore the memory to the guest, the hypervisor can deflate the balloon by mapping new memory for the guest and giving it to the balloon driver, which frees it back to the OS. In general, this leads to minimal disruption of the guests, as the balloon driver has control over what memory to return to the hypervisor.

However, this inflation/deflation process only works within the bounds on the amount of physical memory that were set when the guest was booted. The balloon driver cannot return physical memory to the free pool of the guest that did not previously exist. To address this limitation, the balloon driver could utilize memory hotplug to enable expansion of the guest beyond its initial physical memory allocation [21]. Effectively, the balloon driver can direct the guest to hotplug additional memory, allocate that memory into the balloon, and then release it to the guest (by freeing it) when directed to do so by the hypervisor. The hypervisor prevents the balloon driver from deflating the balloon when it is not supposed to by controlling the mappings in the hypervisor page tables. However, to inflate the balloon, the hypervisor needs the balloon driver to provide free pages that are safe to unmap.

While Xen currently has ballooning capabilities, guest memory is not reallocated unless directed by a system administrator using command line utilities. There are no tools distributed with Xen to actively monitor the system and dynamically adjust the memory allocation among guests.

2.2 Xenstore

The xenstore is a simple key-value store that serves as a communication channel between Xen and its domains. Depending on permissions, a given key can be written to by guest domains and/or Domain 0. Likewise, both guests and Domain 0 can register a watch on a key, subscribing for notification of all writes to that key.

Stock Xen uses two keys to describe a guest's memory reservation: a `target` and a `static_max`. The `target` can be manually increased or decreased at runtime, triggering deflation or inflation of the guest's balloon, respectively. The `static_max` is a fixed upper bound on the guest's memory reservation, and the `target` can be set to any value less than or equal to the `static_max`. This paper proposes to invert the existing schema, removing the upper bound and replacing it with a fixed lower bound, `static_min`. Each guest is guaranteed its minimum reservation at all times, even if its current memory utilization is low. However, as Sections 3 and 4 will describe, a guest may be granted additional memory beyond its `static_min` if it faces high memory pressure. In situations where memory is over-committed, the guests' `static_min` values play a role in computing a fair distribution of memory amongst guests.

2.3 Hypervisor Paging

Given that the hypervisor cannot force the balloon driver to inflate its balloon, the obvious next step is for the hypervisor to swap out guest pages when the balloon driver does not comply with its requests [1, 22].

Xen ships with a simple, experimental page daemon, `xenpaging`, that runs in Domain 0. It takes a domain and a fixed target size as input, and keeps the domain at that size by paging out whenever the domain's memory allocation exceeds its target. Stock xenpaging uses a basic sequential policy when selecting pages for eviction: it simply scans the guest physical address space sequentially until it finds enough pages to evict. When a page fault occurs to a page that has been swapped out, the hypervisor sends a page-in event to xenpaging. xenpaging then pages in the requested page and notifies the hypervisor.

In contrast to a conventional paging system, xenpaging does not attempt to keep a buffer of free pages for a domain in order to allow page-ins to occur before evictions. Instead, it relies on there being additional memory available that is not allocated to any domain for demand page-ins, and then it evicts pages from the domain after it has already exceeded its allocation. If there is no available memory, then xenpaging blocks forever waiting until free memory becomes available (rather than evicting pages first).

For this work, xenpaging has been updated to support multiple guests, allow the target sizes to change, use a random eviction policy (as suggested in [22]), opportunistically page in pages when the target size increases, and never block waiting for free memory. Additional optimizations, as suggested in [1], could be performed, but they are orthogonal to this work.

3 MECHANISMS

This section presents the mechanisms used to implement dynamic, demand-based reallocation of memory. The system architecture, shown in Figure 1, includes components within Domain 0, the hypervisor, and the guest domains. The bulk of the implementation is contained within flexmemd, a new daemon in Domain 0 that dynamically allocates memory. However, mechanisms have also been added to the guest's memory management subsystem to achieve two goals. First, flexmemd must understand the guests' current demands in order to set their targets intelligently. While it could monitor and measure the guests, it currently relies on direct communication from paravirtualized guests. Second, guests must be able to efficiently meet the targets set by flexmemd. Section 3.1 describes the intra-guest mechanisms used to accomplish these two goals. Section 3.2 then outlines flexmemd's approach to handling guests' demands, focusing on its policy-agnostic mechanisms. The specific policies that have been implemented on top of these mechanisms will then be discussed in Section 4.

3.1 Intra-Guest Mechanisms

Guests communicate their memory demands to flexmemd so that it can assign appropriate targets; guests subsequently expand or contract to meet these changing targets. Thus far, two highly disparate operating systems, Linux 4.4 and OSv, have been modified for compatibility with flexmemd, demonstrating its ability to support a diversity of guest operating systems.

3.1.1 Context: Guest OSes. The mechanisms that were added to Linux are applicable to other conventional operating systems such as FreeBSD. In contrast, OSv was uniquely designed for the cloud [10]. It cannot run directly on hardware, but rather must run as a guest within a virtualization system, such as Xen or KVM. OSv is a library operating system: a single application is compiled directly into the kernel, producing a single-function virtual appliance.

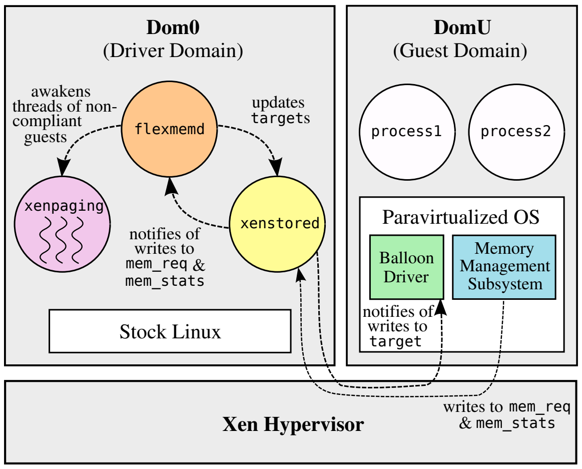
Dom0
(Driver Domain)
awakens threads of non-compliant guests
flexmemd
updates targets
xenpaging
xenstored
notifies of writes to mem_req & mem_stats
Stock Linux
DomU
(Guest Domain)
process1
process2
Paravirtualized OS
Balloon Driver
Memory Management Subsystem
notifies of writes to target
Xen Hypervisor
writes to mem_req & mem_stats

Figure 1: System Architecture Diagram

One key element of OSv is its memory shrinker. OSv does not support memory paging to over-commit physical memory. Instead, the application is responsible for deciding what memory to free by registering a shrinker. When free memory falls below a low memory watermark, OSv calls the registered shrinker with the amount of memory that must be freed to rise above the low memory watermark. If the shrinker does not comply and the system cannot satisfy future allocations, then the entire OSv guest will crash with an out-of-memory error.

3.1.2 Communicating Demands. Guests may communicate demands to flexmemd implicitly — by providing statistics including the number of swaps and amount of free memory — and explicitly — by requesting a specific change in the target. Both are relayed through the xenstore, via two new keys: mem_stats and mem_req, respectively. Which of these flexmemd uses to compute guests' new targets depends on the specific policy configuration, as will be described in Section 4.

Statistics must be sent periodically; both Linux and OSv have been modified to send statistics once per second. While malicious guests could send false information, flexmemd offers protection against this in the form of a credit system that will be described in Section 3.2.3. Furthermore, this paravirtualized approach could be replaced by modifying the virtualization system to measure and infer these statistics.

In addition to sending statistics periodically, guests may send emergency statistics indicating extreme memory pressure. Receipt of an emergency will trigger immediate execution of flexmemd's allocation policy, bypassing any wait period the policy may have. However, declaring an emergency does not guarantee that a guest will get more memory; that decision will be made by flexmemd according to its usual policies. Moreover, flexmemd will respond to only one emergency per guest per policy period to prevent guests from taking advantage of this capability. Linux has been modified to declare emergencies if it substantially struggles to reclaim memory. Specifically, in Linux, allocation or swapping may cause the guest to scan its memory and attempt to reclaim pages; if more than 60%

of its attempts to reclaim pages during the scan fail, it declares an emergency.

Further, both Linux and OSv have been modified to send explicit requests. Linux requests additional memory just before waking up the kernel swap daemon (kswapd). However, it only allows one outstanding request at a time. This serialization is mutually beneficial. It prevents flexmemd from being flooded with requests, while also ensuring that the guest does not waste its credits on redundant requests, since in many cases granting the request that is currently in flight will be sufficient to alleviate its memory pressure.

OSv has been modified to request additional memory before invoking the shrinker. OSv does not perform migration to coalesce free memory, so if memory becomes highly fragmented, large allocations may fail even if there is sufficient total free memory. Ideally, OSv would be modified to migrate memory in order to coalesce more free memory when it finds itself under such conditions, but that is out of the scope of this project; in the interim, it has been modified to request more memory from flexmemd instead.

Guests may also send "negative" requests to indicate willingness to release unneeded memory. Once again, the credit system motivates good citizenship: releasing memory in periods of low pressure allows the guest to save credits for future periods of high pressure. Linux sends negative requests when its ratio of active to free and/or inactive memory exceeds (0.8 * inactive_ratio), where the inactive_ratio is an existing value representing the target ratio of active to inactive anonymous memory on the LRU. Note that Linux increases the inactive_ratio with the size of the guest's memory reservation, such that guests demand a lower proportion of free memory when they have more total memory.

In contrast, OSv does not have a notion of inactive memory, and its only target with respect to free memory is the low watermark. Thus, a new high watermark has been added, set based on empirical measurements. OSv begins issuing negative requests when free memory rises above 25% of the total memory.

3.1.3 Expanding & Contracting. Dynamically changing guest memory reservations is accomplished via ballooning. As described in Section 2.1, Linux includes a balloon driver with memory hotplug support. However, the stock balloon driver is designed to minimize the impact on the guest operating system, rather than to minimize the time taken to respond to a new target allocation. As a side effect, the balloon inflates and deflates relatively slowly. This is not unique to the Xen balloon driver, but is rather the general structure of such drivers (across operating systems and virtualization systems). When the memory allocated to a guest is being actively managed, it is critical for the guest to react more quickly, especially in scenarios where there may be large changes in a guest's memory reservation. Increased reactivity allows expanding guests to take advantage of additional memory as soon as possible, and minimizes the chances of contracting guests being paged out by the hypervisor.

To enable guests to react more quickly, the structure of the balloon driver was redesigned and optimized. First, the use of delayed work has been improved to ensure that the balloon driver awakens more promptly to meet flexmemd's demands. Second, the allocation flags used during inflation have been made more aggressive. Prompt voluntary compliance is mutually beneficial. Clearly, it benefits other guests who are waiting to receive memory. Perhaps less intuitively, it also benefits the guest whose balloon is being inflated, due to flexmemd's use of the xenpaging daemon to enforce compliance. This will be described in greater detail in Section 3.2.2.

Third, the hotplug mechanism in the balloon driver has been modified to allow guests to ramp up their memory reservations more smoothly. The time it takes to enable hotplugged memory is proportional to the size of the memory being added. The original mechanism hotplugged memory in a single chunk, requiring the guest to wait for the entire amount to be added before using even a small fraction of it. This has been changed to an incremental process in which memory is added in chunks no larger than 512 MB, allowing the guest to begin accessing its new memory sooner.

Further, the original mechanism waited to hotplug memory until the balloon was completely deflated, and then hotplugged exactly the amount needed to get to its target. For efficiency, the balloon driver has been modified to preemptively hotplug memory before the balloon is completely deflated. If the balloon is deflating and the target will ultimately require hotplugging memory, then the hotplug is initiated once there is less than 256 MB in the balloon. Moreover, extra memory (capped at 512 MB) is hotplugged at that time, in anticipation of future increases. These thresholds and limits were determined empirically based on the behavior of the Linux kernel. Note that the guest does not actually gain the use of more memory than it was assigned by flexmemd. Rather, the balloon is kept partially-inflated, preventing the guest from accessing the excess while facilitating more rapid expansion in the likely event that the guest continues to require additional memory. As will be discussed in Section 5.4, these modifications improved the rate of memory transfer between guests by an average of 51.5%.

Unlike Linux, OSv did not originally support hotplugging or ballooning; both have been implemented using a similar state machine that includes the optimizations described here. Support for hotplug primarily consisted of modifying the virtual memory system to recognize that the amount of memory is no longer static.

Ballooning began as a port of FreeBSD's balloon driver, but required substantial changes to coordinate with OSv's memory management subsystem, which is quite different from that of a conventional operating system. In particular, the balloon driver must coordinate with the shrinker. When flexmemd decreases the guest's reservation, the balloon begins inflating, creating memory pressure. This pressure awakens the shrinker, which subsequently attempts to shrink the applications; at this point, the balloon driver allows the shrinking process to proceed before inflating the balloon further, leaving memory for the shrinker itself to function. When flexmemd increases the guest's reservation, the balloon begins deflating. During this process the shrinker is disabled, as any pressure during this period is only temporary. When the balloon driver finishes deflating the balloon, it notifies applications that are waiting for allocations to retry.

Memory hotplug is an integral part of the system. Theoretically, regardless of its actual allocation, each guest could be told it has all of the memory in the machine. Any excess beyond its initial target could then be used to inflate the balloon. Later, the guest could inflate/deflate the balloon as its allocation changes. However, many of the overheads involved with tracking memory are proportional to the amount of physical memory. A virtualized system could easily have orders of magnitude more memory than is allocated to any one guest, wasting a large fraction of memory on useless bookkeeping. For example, in Linux on a machine with 1 TB of memory (not unreasonable for a large-scale server), the page structures needed to track all memory alone would consume 8 GB. This is just one of the overheads which would force much larger allocations than would be desired for virtual machines with relatively small workloads. Using hotplug is more flexible and efficient than simple ballooning, as it keeps the overheads proportional to the actual memory the guest has been allocated. With hotplug, a virtual machine could expand to consume the entire TB without requiring additional bookkeeping overhead until it is actually necessary. Furthermore, a guest could migrate to a machine with a larger amount of memory, in which case hotplug would be required to take advantage of it.

3.2 `flexmemd`

While the guests execute their workloads, the `flexmemd` daemon awaits notification of their changing demands. It watches each guest's `mem_stats` and `mem_req` keys in the xenstore, and responds to changes by reallocating memory intelligently and fairly according to one of several configurable policies (see Section 4). Though `flexmemd` supports a variety of policies, these policies share a core set of mechanisms used to monitor demand, compute targets, manage credits, and enforce compliance.

At a high level, `flexmemd` begins by initializing a set of internal structures, establishing connections to Xen and the xenstore, and watching the `mem_req` and `mem_stats` keys. It then sits in an event loop awaiting guests' demands. Algorithm 1 sketches an outline of `flexmemd`'s workflow; Sections 3.2.1–3.2.3 describes the key mechanisms that implement this workflow.

Algorithm 1: `flexmemd` event loop

```
Input: argv, argc; command-line arguments specifying the
       allocation policy
Output: none
/* Extract policy specification and allocate guest */
/* metadata structures.                            */
1  $policy, metadata \leftarrow$ flexmemd_init($argv$, $argc$);
2  $q \leftarrow$ empty queue;
3  while true do
       /* Receive and enqueue new requests; detect   */
       /* emergencies (e).                            */
4      $e \leftarrow$ await_xenstore_events($q$);
       /* Periodically handle tasks such as          */
       /* taxation and paging.                        */
5      $curr \leftarrow$ current time;
6      foreach $pe \in$ periodic_events($policy$) do
7          if $period\_end_{pe} \geq curr$ then
8              $event\_handler_{pe}()$;
9              ...
       /* Determine whether it's time to             */
       /* reallocate.                                 */
10     if (is_periodic($policy$) $\wedge$ $curr < period\_end$ $\wedge$ $\neg e$)
       $\vee$ (is_immediate($policy$) $\wedge$ $q$ is empty) then
11         continue;
12     ...
       /* Partially refund credits if previous       */
       /* rental period is cut short.                 */
13     if $e$ then
14         refund_credits($metadata$);
       /* Execute allocation policy.                  */
15     $targets \leftarrow policy\_fxn\_table[policy](q)$;
       /* Write new targets to the Xenstore.          */
16     set_xenstore_targets($targets$);
```

3.2.1 Computing Availability. To allocate memory amongst the guests, `flexmemd` must first know how much memory is available. It tracks three notions of available memory: physical (P), free (F), and rentable (R). Physical memory is self-explanatory. The values of free and rentable memory depend on a configurable amount of memory that is reserved for use by the hypervisor (H), as well as the current system memory configuration — namely, the `static_min` and `target` values of each domain d in the set of guest domains D (where D excludes Domain 0). Specifically, F and R are defined by the following equations:

$$F = P - H - \texttt{target}_{dom0} - \sum_{d \in D} (\texttt{target}_d) \quad (1)$$

$$R = P - H - \texttt{target}_{dom0} - \sum_{d \in D} (\texttt{static\_min}_d) \quad (2)$$

Thus, F represents memory that is not currently reserved for either the hypervisor itself or any of the domains, while R represents the subset of memory that can be dynamically reallocated to any guest. Each guest is always guaranteed its `static_min`.

3.2.2 Paging. As described in Section 2.3, `xenpaging` was originally designed as a stand-alone utility to enforce a manually-specified, fixed target for a single guest. It has been re-factored into a subsidiary of `flexmemd`, which creates one `xenpaging` thread per guest. These threads initially sleep, waiting for `flexmemd` to provide a new target number of pages.

As `flexmemd` executes its policies, it computes new targets for the guests and conveys these to the hypervisor via the xenstore; the hypervisor then activates the guests' balloon drivers to meet these targets. If a guest's new target is less than its previous target, `flexmemd` does not immediately awaken `xenpaging`. Rather, it first gives the guest's balloon driver a chance to respond. Eventually, `flexmemd` may deem the guest non-compliant, at which point it will awaken that guest's `xenpaging` thread to evict random pages until the target is reached. It is therefore in the guests' best interests to cooperate with the balloon driver by paging themselves out, as this allows them to choose a more favorable set of pages to evict.

If a guest's new target is greater than its previous target, flexmemd awakens its xenpaging thread immediately to begin paging in any pages that were previously paged out. Note that the original xenpaging code would only page in pages on demand. By paging in preemptively as memory is expanded, flexmemd minimizes unnecessary page faults and performance loss.

3.2.3 Managing Credits. To enforce fairness, flexmemd includes a credit system in which guests must spend credits to rent memory. Each guest is given an initial allocation of credits proportional to its static_min. At runtime, the price of memory varies based on supply and demand. When the system is under-committed, memory is free; as demand exceeds supply, the price rises in accordance with the specific policy being used (see Section 4). Policies are not required to use this credit system, though most of the policies implemented thus far do.

Guests are periodically charged for memory consumption beyond their static_min. Spent credits are redistributed amongst the guests, again proportional to static_min values to preserve fairness. Credits are also periodically reallocated via a savings tax. Every second, 5% of each guest's credits are redistributed proportionally amongst the guests. This prevents guests from hoarding credits indefinitely. Finally, if an emergency disrupts the scheduled period, guests are issued partial refunds in proportion to the amount of the period that was remaining when the emergency occurred.

4 POLICIES

This section describes the policies that are currently supported by flexmemd. Designed for extensibility, flexmemd supports a wide variety of allocation policies. Policies are composable, and are characterized by four key features: frequency of execution, algorithm for setting the price of memory, algorithm for computing guests' desired targets, and allocation algorithm (a function of desired targets and prices). flexmemd offers multiple options for each of these, and users can specify the desired configuration at the command line.

In terms of frequency, a policy may execute only once, respond immediately to each xenstore event, or periodically process batches of events. The price of memory may be set to a fixed rate, scaled according to the proportion of rented memory a guest already holds, or set using a custom formula. Desired targets can be computed based on explicit requests (mem_req) or implicit requests (mem_stats). The former simply consists of adding mem_req, a delta, to the guest's current reservation.

For configurations that use implicit requests, flexmemd performs several steps to infer each guest's demand. It first computes a desired percent free (dpf) for each guest based on that guest's current reservation ($\texttt{old\_target}_d$) according to the following equation:

$$\texttt{dpf}_d = \text{MIN}(0.25, \frac{1}{sqrt((9 * \texttt{old\_target}_d) + 1)}) \tag{3}$$

Note that this reserves a larger proportion of free memory when the guest's reservation is smaller. Next, flexmemd examines the guest's swap rate and amount of free memory to determine whether change is needed. flexmemd uses configurable thresholds for each of these — by default, 90 swaps/s and 100 MB of free memory. If the guest's swap rate exceeds the swap threshold or its free memory falls below the free memory threshold, flexmemd declares that the guest desires an increase. Conversely, flexmemd determines that a decrease is desirable if the guest's swap rate is below the swap threshold and its percentage of free memory exceeds dpf. Otherwise, if the guest is not swapping heavily and has less free memory than dpf but more than the 100 MB threshold, no change is made to the guest's desired target. In both the increase and decrease conditions, a new desired target is computed by solving the following equations to ensure that the guest has 2% more free memory than dpf:

$$\texttt{delta}_d = \frac{((\texttt{old\_target}_d * (\texttt{dpf}_d + 0.02)) - \texttt{free\_mem}_d)}{(1 - \texttt{dpf}_d + 0.02))} \tag{4}$$

$$\texttt{new\_target}_d = \texttt{old\_target}_d + \texttt{delta}_d \tag{5}$$

Finally, the most fundamental differentiator between policies is the allocation algorithm. Sections 4.1–4.6 describe six allocation algorithms that have been implemented in flexmemd thus far. With the exception of PROPORTIONAL, each of these allocation algorithms is compatible with multiple combinations of the aforementioned three features, for a total of 16 supported configurations. For simplicity, a single evaluation configuration was chosen for each, as will be described in the subsequent sections.

4.1 Proportional

The PROPORTIONAL policy is static and demand-agnostic; it simply allocates rentable memory to guests in proportion to their static_min values. As evaluated in Section 5, it executes only once, makes no attempt to ascertain the guests' desired targets, and does not use credits. While this policy offers strict fairness guarantees, its inability to adapt to demand can lead to under-utilization of memory and performance degradation, as will be shown in Section 5.

4.2 First-Come First-Serve

The FCFS policy improves adaptability by taking demand into account: as it receives requests from the guests, it allocates on a first-come, first-serve basis. It attempts to maintain the status quo, subject to guests having sufficient credits. The configuration used in the evaluation responds immediately to explicit requests, and sets the price in proportion to the requester's current rented memory.

Unlike PROPORTIONAL, this policy never allocates memory to a guest who has not expressed a need for it, which prevents waste at the outset; negative requests then eliminate waste as demands change. The credit system and proportional pricing policy prevent guests from hoarding memory in the long-term, but allows them to temporarily hold more than their proportional share if, for instance, the supply of rentable memory exceeds the aggregate demand or if they have amassed extra credits due to renting less in the past.

4.3 Demand-Based Proportional

DEMAND_PROP attempts to combine the rigid fairness of PROPORTIONAL with the superior utilization of FCFS. Like FCFS, it will not give memory to a guest unless that guest requests it.

As long as the supply exceeds the total demand, Demand_Prop behaves identically to FCFS: it maximizes utilization by allowing guests to rent as much memory as they desire, even if this is more than they deserve relative to their `static_min` values. However, as demand exceeds supply, Demand_Prop ensures that each guest always receives at least the minimum of its desired target ($\texttt{desired\_target}_d$) and its fair proportion of rented memory ($\texttt{static\_min}_d + (R \times \frac{\texttt{static\_min}_d}{\sum_{d \in D}(\texttt{static\_min}_d)})$). If its desired target exceeds its fair proportion, it may be granted an intermediate value, depending on the demands of the other guests; however, if its fair proportion exceeds its desired target, it will not be granted more than MAX($\texttt{desired\_target}_d$, $\texttt{static\_min}_d$). This ensures that memory is not wasted needlessly.

The configuration of Demand_Prop used in the evaluation responds immediately to explicit requests. This causes `flexmemd` to set the desired target for the guest that issued the most recent request to its current target ($\texttt{old\_target}_d$) plus its requested value ($\texttt{mem\_req}_d$), while all other guests' desired targets are simply their current targets.

Like Proportional, Demand_Prop does not use a credit system, as it inherently enforces fairness. It uses an iterative algorithm centered around the notion of classifying guests as greedy if they are currently renting more memory than they deserve relative to their `static_min`. If `flexmemd` receives a request of size `req` from a guest who holds a smaller proportion of rented memory than one or more greedy guests, it will take memory from the greedy guest(s). When the re-balancing process terminates, the requester's reservation will have been increased by either `mem_req` or the amount needed to tie its proportion of excess, whichever is smaller. Negative requests enable it to adapt to decreases in demand, avoiding the wasteful equilibrium of Proportional.

4.4 Direct Assign

The Direct_Assign policy directly computes a custom memory price based on guests' demands and credits. It was evaluated periodically using implicit requests; thus, it first computes each guest's desired target using Equations (3)–(5). If supply exceeds the total demand, the price is set to zero credits per KB. This ensures that when memory is plentiful, guests are never prevented from reaching their desired targets. However, if demand exceeds supply, Direct_Assign computes an integer price per KB that allows the guests to jointly consume all of the rentable memory (R), but would not allow them to consume more memory than is available. More formally, it searches for the maximum price p that satisfies the equation $\texttt{tot\_rented}(p) \leq R$, where:

$$\texttt{rented}_d(p) = \begin{array}{r}\text{MIN}(\texttt{desired\_target}_d - \texttt{static\_min}_d,\\ (\texttt{credits}_d/p), \texttt{avail\_mem})\end{array} \tag{6}$$

$$\texttt{tot\_rented}(p) = \sum_{d \in D}(\texttt{rented}_d(p)) \tag{7}$$

This is an iterative process. First, `avail_mem` is initialized to R. Next, $\texttt{rented}_d(p)$ is computed for each guest d in order of descending $\texttt{credits}_d$. After processing the nth guest d, `avail_mem` is decremented by $\texttt{rented}_d(p)$ before the $n + 1$th guest is processed.

Having computed p, Direct_Assign then allocates $\texttt{rented}_d(p)$ to each guest.

4.5 Auction

The Auction policy allocates memory amongst guests using a sealed-bid auction. Like Direct_Assign, its evaluation configuration periodically handles implicit requests using a custom pricing algorithm. Once again, Equations (3)–(5) are used to intuit demand, and renting memory costs nothing if total demand is low. However, if demand exceeds supply, Auction constructs a bid for each guest. A guest's bid is computed as the maximum amount it can pay while still being able to afford its desired target, according to the following equation:

$$\texttt{bid}_d = \begin{cases} \dfrac{\texttt{desired\_target}_d - \texttt{static\_min}_d}{\texttt{credits}_d}, & \\ \quad \texttt{desired\_target}_d > \texttt{static\_min}_d; & \\ 0, \quad \texttt{desired\_target}_d \leq \texttt{static\_min}_d; & \end{cases} \tag{8}$$

Last, the Auction policy simulates an auction by choosing the highest $\texttt{bid}_d$ that will allow all of the memory to be rented. Rentable memory is then allocated amongst the guests in descending order of bids.

4.6 Round Robin

Round_Robin is a variant of the Auction policy that was motivated by the observation that, when demand exceeds supply, the Auction policy tends to oscillate frequently. Consider a system with 4 GB of rentable memory and two guests, each of whom have a similar number of credits and would happily consume the entire 4 GB. The guest that currently has more credits — even if only by a slim margin — will be granted as much of this 4 GB as it can afford. After a period of time, this will result in the balance shifting such that the other guest has more credits, and will now receive the bulk of the memory. While long-term fairness is maintained, constantly shifting memory between guests can be unnecessarily disruptive.

Therefore, although Round_Robin uses the same bidding scheme as Auction, it adapts the algorithm by splitting allocation into rounds. The maximum amount that a guest d may receive in a round is configurable; it can be set to either 1 GB, or 0.25 * $\texttt{static\_min}_d$, where $\texttt{static\_min}_d$ is in GB. For evaluation, the fixed limit of 1 GB was used. Note that a guest may be granted less if it is out of credits or its demand has already been satisfied in prior rounds. The algorithm progresses until all of the memory has been allocated or all of the guests have been satisfied. For evaluation, Round_Robin was configured to periodically react to implicit requests.

5 EVALUATION

This section first describes the methodology, microbenchmarks, and applications used to evaluate the effectiveness of `flexmemd` and its policies. It then examines the performance and fairness of `flexmemd`'s dynamic allocation policies through a variety of experiments involving 1–3 guests.

Table 1: Application Performance: Single Guest

Policy	Linux: Throughput (ops/s)			OSv: Hit Rate
	Memcached	PostgreSQL	GraphChi	Memcached
Proportional	62636	414.0	10397324	100%
FCFS	61946	140.8	9504965	91.5%
Demand_Prop.	53864	144.9	9434685	91.0%
Direct_Assign	59254	236.9	10043572	99.9%
Auction	55077	238.0	9995738	99.7%
Round_Robin	57290	235.1	10283591	99.9%
Static allocation	Timeout	121.2	4900517	71.3%

5.1 Methodology

All experiments were run on a system with an Intel Xeon E3-1231 v3 processor and 32 GB of DDR3-1600 memory. The system ran Xen 4.6, with its driver domain utilities modified as previously described. Domain 0 was configured to use up to 4 virtual CPUs, have a fixed 4 GB memory allocation, and run Ubuntu 14.04 LTS using Linux 4.2. It had access to a 250 GB SSD, which `xenpaging` used as swap space for the virtual machines. An additional 4 GB was reserved for the hypervisor; this was an over-allocation that was more than sufficient to ensure space for all hypervisor data structures. This left the remaining 24 GB in the system for use by the virtual machines.

Each guest ran a minimal installation of Ubuntu 14.04 using Linux 4.4 or OSv, modified as described in Section 3. Guests were given 32 GB LVM disk partitions on a single 1 TB hard drive. Each experiment was run with freshly booted virtual machines.

The system was evaluated using three applications: Memcached, PostgreSQL, and GraphChi [13]. The workloads for each application were configured to push guests towards a memory footprint of roughly 8–12 GB. Memcached was configured with a 10 GB cache and was exercised using memaslap with a 90% read workload. Pgbench was run on a PostgreSQL database with a scale factor of 300, with 5 clients making 500 transactions each. GraphChi was used to run the PageRank algorithm [19] on a Twitter graph provided by the Stanford Large Network Dataset Collection [14]. Although Linux was evaluated with all three applications, OSv was only evaluated with OSv's specially modified version of Memcached. This version of Memcached integrates with OSv's shrinker to dynamically expand and contract the cache to utilize memory efficiently. The use of other applications would require porting them for compatibility with OSv's shrinker. Additionally, two microbenchmarks were used to provide controlled use of memory within a guest. `constant` maintains a fixed working set size, while `flux` oscillates between two constant working set sizes.

5.2 Isolated Application Performance

Each application was first run in isolation in a single guest with no other guests running on the system. This shows how `flexmemd` reacts to these applications' demands when there is no contention for memory. In each case, the guest began with an initial reservation of 4 GB.

Table 1 shows the performance results of the applications when using the different allocation policies, as well as a comparison to a static allocation of 4 GB. The results for Linux guests are reported as application throughput. For Memcached, Linux achieved a 100% hit rate in all experiments. However, the OSv shrinker forces data to be evicted, which leads memaslap to over-report throughput (misses are faster than hits, since memaslap does not go to disk on misses). So, for OSv, the hit rate within the memcache is a better indicator of Memcached performance.

Given that there was only a single guest running, the Proportional policy allocated all of the rentable memory to that guest. Since none of the applications required the full 24 GB, their performance under the Proportional policy approximates their best-case performance. On the other hand, leaving the guests at their initial static allocation of 4 GB caused the applications to become overwhelmed and face massive performance degradation due to memory starvation.

The remainder of the table shows that the five dynamic allocation policies all reacted quickly enough to assign additional memory to the guest when the applications required it. In particular, they reacted quickly enough such that all policy/application configurations achieve performance within 14% of the ideal (Proportional), with the exception of PostgreSQL. PostgreSQL heavily utilizes the file buffer cache within the OS. Under memory pressure, the OS quickly reclaims clean buffer cache pages, even as `flexmemd` allocates additional memory, resulting in lower hit rates. While this led to lower performance in this particular case, it would likely be worse in the general case for the operating system to allow the buffer cache to grow without bound when there is memory pressure.

There were a few small variations in how the different applications in isolation responded to the various policies. In general, Direct_Assign and Round_Robin achieved the best performance, though FCFS also performed well for Memcached on Linux. The most stark difference was seen in PostgreSQL, where the two request-based policies (FCFS and Demand_Prop) suffered significantly. As will be shown in Section 5.4.1, these request-based policies tend to maintain tighter bounds on the memory that is granted to a guest than the statistics-based policies, causing them to be far more sensitive to the quick reclamation of the buffer cache.

5.3 Multi-Guest Performance

Table 2 shows application performance when there are multiple guests on the system, one running an application and up to three others running the `constant` microbenchmark. The results for these experiments were reported for both an under-committed

Table 2: Application Performance: Multiple Guests

Policy	Linux: Throughput (ops/s)						OSv: Hit Rate	
	Memcached		PostgreSQL		GraphChi		Memcached	
	Under	Over	Under	Over	Under	Over	Under	Over
Proportional	36490	Timeout	184.4	166.8	10039406	5048317	97.8%	96.8%
FCFS	53818	Timeout	132.4	141.1	9366516	8784397	92.2%	84.2%
Demand_Prop.	59346	54400	128.4	122.6	9320775	9165946	91.9%	91.3%
Direct_Assign	53143	36154	196.8	161.3	8620764	7222453	99.9%	96.0%
Auction	58306	35475	228.5	154.0	9087851	5216996	100%	95.7%
Round_Robin	50330	27137	222.2	156.6	9030675	5721226	99.9%	97.5%

Table 3: Convergence Fairness on Linux (GB)

Policy	(10, 10)		(10, 20)		(20, 20)	
	VM1	VM2	VM1	VM2	VM1	VM2
Proportional	12.0	12.0	12.0	12.0	12.0	12.0
FCFS	10.4	10.3	10.1	13.9	11.0	13.0
Demand_Prop.	10.5	10.3	10.1	13.9	12.0	12.0
Direct_Assign	11.4	10.7	10.7	13.3	12.0	12.0
Auction	11.3	11.5	10.3	13.7	12.4*	11.0*
Round_Robin	11.6	10.7	10.4	13.6	12.0	12.0
Expected	≥10.0	≥10.0	≥10.0	≤14.0	12.0	12.0

case ("Under") and an over-committed case ("Over"). In the under-committed case, the applications were run alongside two guests whose constant working set sizes were 4 GB each. For Linux, the over-committed case consisted of three additional guests with constant working set sizes of 4.5 GB each; for OSv, the over-committed case consisted of two additional guests with constant working set sizes of 8 GB each. These working set sizes are a lower bound on what the guests will actually consume, as they will attempt to claim additional memory to support kernel processes and to serve as a buffer in case of future growth. In general, `flexmemd` grants roughly an additional 0.5–1.5 GB to each constant guest. All of the applications ideally require 8–12 GB of memory, so in the over-committed case they must compete with the `constant` guests.

The first thing to note is that Proportional no longer necessarily gave the guests all of the memory they needed. If a guest could profitably use more than its fixed fraction of the total memory, then the application's performance could degrade substantially, as occurred in both the under- and over-committed cases of Memcached on Linux, as well as in the over-committed case of GraphChi. Second, the table further highlights the need for dynamic memory reallocation: it shows that all of the policies enabled the applications to out-perform the fixed, static allocation from Table 1, in both the under- and over-committed cases.

Finally, this table reveals the relative merits, in terms of application performance, of the different policies when there is memory contention. For PostgreSQL on Linux and Memcached on OSv, Auction and Round_Robin typically won by intelligently purchasing memory on behalf of the application guest. On PostgreSQL in particular, FCFS and Demand_Prop were once again thwarted by the OS' quick cleanup of the buffer cache. This, combined with the poor performance of FCFS on Memcached, points to a broader issue with these request-based policies: unpredictability as a result of their sensitivity. They react quickly to small fluctuations in the amount of free memory, which often prevents them from growing sufficiently large to support their workloads. However, for GraphChi and Memcached on Linux, Demand_Prop typically performed the best. This is primarily an artifact of the tighter bounds on memory maintained by the request-based configurations.

5.4 Fairness

Tables 3 and 4 show the fairness that `flexmemd`'s allocation policies achieve in a variety of different two-guest scenarios. Fairness was evaluated in terms of both the size of the memory reservations each guest received upon converging to an equilibrium point, as well as the time it took to converge to an equilibrium point when the guest's working set sizes were abruptly changed.

5.4.1 Equilibrium Points. Table 3 shows the equilibrium points reached by the policies when two guests executed different constant workloads. The top row of the table shows the workloads of the two guests, (X, Y), where X and Y are the working set sizes, in GB, of VM1 and VM2, respectively. Results are shown for Linux; since fairness is an artifact of `flexmemd`'s policies rather than the intra-guest mechanisms, OSv's results are omitted for brevity.

In the under-committed case of $(10, 10)$, one would expect all of the policies to allocate at least 10 GB to each guest. As there is excess memory, the more aggressive policies might allocate additional memory in anticipation of further growth of the guests. As the table shows, all of the policies allocated roughly the same amount to both guests, and always allocated more than the desired 10 GB. As alluded to previously, the desired percent free equation (3), used by the statistics-based policies (Direct_Assign, Auction, and

Table 4: Convergence Rate (MB/s)

Policy	Linux			OSv	
	(6 ↔ 10)	(10 ↔ 20)	(10 → 20, 20)	(6 ↔ 10)	(8 ↔ 20)
Proportional	N/A	N/A	N/A	N/A	N/A
FCFS	124.9	41.5	37.1	175.2	213.1
Demand_Prop.	131.6	37.1	37.0	176.8	337.9
Direct_Assign	349.7	36.3	40.1	46.9	37.9
Auction	462.7	27.2	*	56.2	74.9
Round_Robin	455.3	33.6	33.9	58.8	63.3

Round_Robin) attempts to grant guests an additional buffer of free memory beyond that which is absolutely necessary.

In the $(10, 20)$ case, memory was over-committed. The two guests started with the same `static_min`, so the smaller guest was able to acquire its full desired allocation of 10 GB and the larger guest then acquired the remainder of the memory. Notably, the smaller guest received less excess memory than it did in the $(10, 10)$ case, as that memory was actually being demanded by the other guest.

Finally, in the $(20, 20)$ case, both guests wanted more memory than they could fairly receive, so the system converged to each guest acquiring roughly half (12 GB) of the total memory. However, the FCFS policy was slightly lopsided, as the first guest to request the final GB gets it — at least until it eventually expends all of its credits. Similarly, the $*$'s in the Auction column indicate that the guests oscillated rather than converging to a static allocation, as explained in Section 4.6.

5.4.2 Convergence Rate. Table 4 shows the rate at which the guests converged to new allocations after `flexmemd` adjusted their working set sizes. Each of these experiments involved two guests. Both guests began by running a microbenchmark with a specific working set size. Then, after the system reached a state of equilibrium, one or more guests was triggered to change its working set size. Once again, the top row of the table denotes the working set sizes of the two guests in GB. Three cases were evaluated on Linux; two on OSv. The two OSv cases were roughly equivalent to the first two cases evaluated on Linux. The third Linux case was omitted for OSv since, as Table 4 will show, the second and third cases behave almost identically in practice. All cases were designed such that the guests exchanged roughly 4 GB of memory.

The first case, (6 ↔ 10), was a scenario in which memory was under-committed. The guests began with working set sizes of 6 GB and 10 GB, and were simultaneously triggered to swap workloads. Since the sum of the two guests' desired memory consumption was less than the total available memory at all times, the initially-smaller guest was free to expand immediately, without needing to wait for the initially-larger guest to release any memory.

In (10 ↔ 20), the sum of the two guests' desired memory consumption was greater than the total available memory at all times. Initially, the smaller guest (VM1) desired 10 GB: more than its `static_min`, but less than it's fair share of the memory. Thus, it was granted all of the memory that it requested. At the same time, the larger guest (VM2), who desired 20 GB, expanded to consume all remaining available memory: approximately 14 GB. As in (6 ↔ 10), the guests were later triggered to simultaneously swap workloads; however, in this case VM1 had to wait for VM2 to shrink before it could expand. Note that the equivalent experiment for OSv used 8 GB rather than 10 GB for the smaller working set size; this is because the interactions between OSv and `flexmemd` typically result in the guest being granted a larger buffer of free memory, particularly as the guest size grows. Therefore, to enabled a transfer of approximately 4 GB of memory, a smaller initial working set size was needed.

The last case, (10 → 20, 20), began with the same configurations as the second case. However, while VM1 was triggered to switch to a working set size of 20 GB, VM2 remained at its initial working set size of 20 GB. This represented the worst possible scenario, as VM1 had to wait for VM2 to release memory against its will.

While all of the policies except Auction ultimately converged to the expected fair allocation of memory, they did so at different rates. The Demand_Prop policy consistently performed best for OSv. For Linux, Auction and Round_Robin performed best in the under-committed case. In general, the statistics-based policies tend to be more sensitive to changes in guests' working set sizes, causing them to expand or contract guests more rapidly. In the over-committed cases, all of the policies performed similarly. This is because when memory is over-committed the rate of convergence is limited by the guests' responsiveness to the balloon driver, which is policy-independent. Note also the similarity in performance between the (10 ↔ 20) and (10 → 20, 20) cases. This indicates that the system is effective in forcing greedy guests to shrink against their will, preserving fairness.

As noted in Sections 4.6 and 5.4.1, the Auction policy fails to converge in certain cases. In particular, when both guests desire more than their fair share of memory, their reservations will fluctuate. This explains the relatively slow convergence rate of Auction in the (10 ↔ 20) case, since there was a period in which both guests desired greater than their fair 12 GB of memory. It likewise accounts for the inability to measure the convergence rate in the (10 ↔ 20, 20) case.

Notably, OSv converged faster in the over-committed case, whereas Linux converged faster in the under-committed case. OSv's behavior is not intuitive, but is a direct result of the shrinker. In the over-committed case, OSv aggressively frees application memory, enabling it to be less busy and respond to increases more quickly. In contrast, Linux must free memory by more slowly swapping to disk when it is over-committed.

Last, to quantify the increased responsiveness of the balloon driver as a result of the changes described in Section 3.1.3, the under-committed experiments were also performed with the stock balloon driver that ships with Linux. Across all policies, the modified balloon driver outperformed the stock balloon driver. On average, it exhibited a 51.5% improvement in convergence rate, with improvements for individual policies ranging from 7.4–154.8%.

5.5 Policy Recommendations

Together, Tables 1–4 tell a complex story about the tradeoffs between policies. Many of the policies' performance and fairness characteristics are driven by whether demand is explicit or implicit. On Linux, the statistics-based policies allow guests to expand more rapidly, resulting in the superior convergence rates seen in Table 4. The statistics-based configurations also tend to anticipate future growth, more so than the request-based policies, and thus are more generous in their estimation of how much memory a guest requires. As a result, they allocate a larger buffer of memory beyond the expected working set size, as shown in Table 3. For instance, in the under-committed case, they allocated an additional 0.4–1.3 GB beyond the tight bounds of the request-based policies.

Comparing amongst the three statistics-based policies presented here, Direct_Assign and Round_Robin exhibit similarly favorable characteristics, though which policy performs best depends on the application. However, the instability of the Auction policy results in less-than-desirable performance and fairness characteristics.

On the other hand, the tendency of statistics-based policies towards over-allocation has implications on performance which are borne out in Tables 1 and 2. Most notably, in cases where the sum of all of the guests' demands only slightly exceeds capacity — as is the case for over-committed Memcached and PostgreSQL on Linux — the applications perform better when using request-based policies. This is because the tighter bounds on the `constant` guests leave more space for the application guest. Thus, if performance is the highest priority, a request-based policy may be desired. On the other hand, the smaller and quicker fluctuations associated with these policies can lead to undesirable behavior. For instance, `flexmemd` may decrease the amount of memory allocated to a guest prematurely in response a momentary change in the guest's working set size. This undesirable behavior is reflected in the results of PostgreSQL with both Demand_Prop and FCFS, and of Memcached with FCFS. Of the two request-based policies evaluated here, Demand_Prop is more successful than FCFS at balancing concerns of performance, fairness, and stability.

6 RELATED WORK

There has been considerable past work on managing resources in virtualized cloud environments across the data center. For example, virtual machine placement can improve the aggregate efficiency of the data center [7, 17]. Resources can also be more efficiently utilized by dynamically increasing or decreasing the number of virtual machines dedicated to a task or service [3, 5, 6, 24]. Furthermore, live migration of virtual machines can simplify data center management and allow dynamic reallocation of physical resources [4, 9, 20]. These higher-level techniques, which operate on the granularity of virtual machines, are complementary to approaches for scaling the resources allocated to a particular virtual machine.

This paper specifically looks at memory allocation among virtual machines within a single physical machine, which has been an important issue in modern virtualization. Past work has attempted to dynamically reassign memory by either equalizing the fraction of free memory among virtual machines [15, 27], estimating the current working set sizes of the virtual machines [22], or predicting the future working set sizes of the virtual machines [11, 23]. However, this past work has focused on how much memory to allocate to a virtual machine within a given, fixed range. These systems do not allocate excess memory efficiently or even allow virtual machines to exceed their fixed size at boot time. In fact, many of the algorithms are explicitly designed around a fixed, maximum allocation per virtual machine. Additionally, much of this work prioritized system-wide memory utilization over fairness to individual guests. In contrast, this paper describes a system in which a large set of diverse policies have been developed to cover a much wider range of scenarios. In particular, the system explicitly focuses on efficiently utilizing excess memory by allowing virtual machines to grow without bound when there is available memory and it is fair and advantageous to do so.

Moreover, the prior work considers very small workloads, typically with footprints of less than 1GB [11, 15, 22, 23, 27]. This is much smaller than common modern workloads, which can consume 10 GB or more of memory. Developing techniques that are specifically tailored towards these larger workloads is important, given the bookkeeping overheads within the operating system and the costs of adding and removing memory from virtual machines. These overheads are proportional to the size of the virtual machine, so small workloads are less affected by these costs. The mechanisms and policies presented in this paper are designed with scalability in mind. This work explores realistic modern workloads on the order of 10 GB, including situations in which virtual machines grow to sizes of two to three times their initial allocation, and deals with the issues involved with such growth. By removing fixed limits and dealing with larger virtual machines, this work has enabled more effective use of large amounts of memory, regardless of whether the host memory is over- or under-committed, the initial sizes of the virtual machines, or how cooperative the guests are.

Past work has also studied the use of auctions to allocate resources, including memory, among virtual machines [2, 12, 18, 26]. In all of this work, customers purchase resources at a fine granularity and short timescales using automated auctions. As the virtual machine has to autonomously bid for its own resources — whose costs will be charged to the customer — past work has necessarily focused on designing agents to decide how much to buy and how much to pay for it at any given time. These agents can be designed to try to optimize for costs, utility, or meeting SLAs. In all of these systems, however, costs must be a primary concern of the agent, as they can vary widely and depend on the other virtual machines present on the system. In contrast, this paper proposes the use of auctions with virtual currency to fairly reallocate memory that would otherwise be idle within the system. Furthermore, the auction takes place entirely within `flexmemd`, so guests need not even be aware of the currency or bidding. This allows `flexmemd` to utilize more innovative and aggressive bidding policies that prioritize fairness and efficiency, without concern for monetary costs (of which there are none) imposed upon the guests.

Transcendent memory enables virtual machines to utilize spare memory that is controlled by the hypervisor [16]. However, memory pages must be copied into and out of transcendent memory using a get/put interface and cannot be accessed directly. Transcendent memory is therefore primarily utilized as an ephemeral clean page cache in which pages are only kept on a best effort basis. The fact that data must be copied into and out of transcendent memory, combined with the fact that pages may be dropped before they are retrieved, limits the overall usefulness to the guest operating system. While guests can acquire persistent transcendent memory under limited circumstances, it is more efficient to allocate persistent memory directly to the guest, eliminating all overheads and restrictions. Thus, while transcendent memory has value, the techniques proposed in this paper provide lower overheads and greater flexibility by allowing the guests direct access to additional memory, while maintaining the ability to fairly utilize spare memory.

Other work has looked at peripheral issues that, while important, do not provide a general solution to memory allocation among virtual machines. Xiong *et al.* explored predicting future resource needs using machine learning techniques [25]. However, their techniques exploit specific characteristics of database systems, and cannot be applied in any obvious way to other workloads. Gupta *et al.* investigated memory management on machines with heterogeneous memory resources [8]. Instead of managing the overall allocation of memory among virtual machines, their focus was primarily on automatically migrating hot pages to faster memory in order to improve performance.

7 CONCLUSIONS

Resource allocation in virtualized systems can have a dramatic impact on the overall performance of the guests running on the system. Memory, in particular, is a critical resource in many modern applications. Relatively few workloads have static memory footprints, so it is essential that the virtualization system react to guests' fluctuating demands. This paper has presented a policy-based system for dynamically scaling the memory reservations of guest VMs.

One key feature of the system is that it allows guests to expand beyond their initial allocations. This enables the system to more effectively use large amounts of memory and accommodate unanticipated bursts in demand, regardless of whether the host memory is over- or under-committed. To accomplish this, the system seamlessly integrates upgraded mechanisms across the hypervisor, driver domain, and guests to implement effective dynamic memory allocation for guest VMs. `flexmemd`, a new daemon that runs in the driver domain and executes a configurable allocation policy, coordinates these mechanisms. Further, as it dynamically reassigns memory amongst the guests, `flexmemd` enforces fairness through techniques such as a virtual credit system.

Several different memory allocation policies were implemented and evaluated within `flexmemd`. Unsurprisingly, there is a tension between fairness and performance. The statistics-based policies are arguably more fair than the request-based policies, as they typically allocate an extra buffer of memory beyond that which the request-based policies allocate. This extra memory serves as a cushion against future growth for guests that currently have low demands, resulting in a more equitable distribution of memory across guests with varied demands. Additionally, in Linux, the statistics-based policies are quicker to respond to changes in demand. Both of these features help achieve fairness by protecting guests whose current demands are low from greedy, memory-intensive guests. In over-committed cases, however, request-based policies can enable memory-intensive guests to more greedily claim memory when it is needed, yielding better performance. Despite this tension, the results indicate that it is possible to achieve a balance between fairness and performance. The DEMAND_PROP and ROUND_ROBIN policies are particularly successful at balancing both fairness and performance across the applications and workloads studied in this paper. Of these, the ROUND_ROBIN policy, which uses VM statistics to transparently execute a virtual auction, is most effective for disk-intensive workloads and OSv. In contrast, the DEMAND_PROP policy, a request-based policy that enforces rigid fairness requirements, is better suited to Linux workloads that have less disk activity.

The system was evaluated with both Linux and OSv, showing that the concepts presented here are largely independent of the guest operating system. Any operating system that can utilize memory ballooning and hotplug in order to change its memory allocation in coordination with `flexmemd` could benefit from the performance improvements provided by the system. As such, the system demonstrates a flexible and general way to improve memory resource allocation for future virtualization systems.

8 ACKNOWLEDGEMENTS

This material is based upon work supported by NSF Award CNS-1619124: "Big Memory: Exploring Memory Management Mechanisms and Policies for Rack-Scale Computers" as well as an NSF Graduate Research Fellowship under Grant No. 1450681.

REFERENCES

[1] Nadav Amit, Dan Tsafrir, and Assaf Schuster. 2014. VSwapper: a Memory Swapper for Virtualized Environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[2] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. 2014. Ginseng: Market-Driven Memory Allocation. In *Proceedings of the 10th ACM SIGOPS/SIGPLAN International Conference on Virtual Execution Environments (VEE '14)*.

[3] Jing Bi, Haitao Yuan, Yushun Fan, Wei Tan, and Jia Zhang. 2015. Dynamic Fine-Grained Resource Provisioning for Heterogeneous Applications in Virtualized Cloud Data Center. In *Proceedings of the 8th IEEE International Conference on Cloud Computing (CLOUD '15)*.

[4] Christopher Clark, Keir Faser, Steven Hand, and Jacob Gorm Hansen. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked System Design and Implementation (NSDI '05)*.

[5] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. 2013. Multi-level Elasticity Control of Cloud Services. In *Proceedings of the 11th International Conference on Service Oriented Computing (ICSOC '13)*.

[6] Rodrigo da Rosa Righi, Vinicius Facco Rodrigues, Cristiano André da Costa, Guilherme Galante, Luis Carlos Erpen de Bona, and Tiago Ferreto. 2016. AutoElastic: Automatic Resource Elasticity for High Performance Applications in the Cloud. *IEEE Transactions on Cloud Computing* 4, 1 (2016).

[7] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*.

[8] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. In *Proceedings of the 11th ACM SIGOPS/SIGPLAN International Conference on Virtual Execution Environments (VEE '15)*.

[9] Michael R. Hines and Kartik Gopalan. 2009. Post-copy Based Live Virtual Machine Migration Using Adaptive Pre-paging and Dynamic Self-ballooning. In *Proceedings of the 5th ACM SIGOPS/SIGPLAN International Conference on Virtual*

Execution Environments (VEE '09).

[10] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*.

[11] Kapil Kumar, Nehal J. Wani, and Suresh Purini. 2015. Dynamic Memory and Core Scaling in Virtual Machines. In *Proceedings of the 8th IEEE International Conference on Cloud Computing (CLOUD '15)*.

[12] Sajib Kundu, Raju Rangaswami, Ming Zhao, Ajay Gulati, and Kaushik Dutta. 2015. Revenue Driven Resource Allocation for Virtualized Data Centers. In *Proceedings of the 12th International Conference on Autonomic Computing (ICAC '15)*.

[13] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 12th USENIX Symposium on Operating System Design and Implementation (OSDI '12)*.

[14] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/. (2014).

[15] Lanzheng Liu, Rui Chu, Yongchun Zhu, Pengfei Zhang, and Liufeng Wang. 2012. DMSS: A Dynamic Memory Scheduling System in Server Consolidation Environments. In *Proceedings of 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW '12)*.

[16] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. 2009. Transcendent Memory and Linux. In *Proceedings of the 2009 Linux Symposium*.

[17] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrious Pendarakis. 2010. Efficient Resource Provisioning in Compute Clouds via VM Multiplexing. In *Proceedings of the 7th IEEE/ACM International Conference on Autonomic Computing (ICAC '10)*.

[18] Dorian Minarolli and Bernd Freisleben. 2011. Utility-driven Allocation of Multiple Types of Resources to Virtual Machines in Clouds. In *Proceedings of the 13th IEEE Conference on Commerce and Enterprise Computing (CEC '11)*.

[19] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford InfoLab.

[20] Shashank Sahni and Vasudeva Varma. 2012. A Hybrid Approach to Live Migration of Virtual Machines. In *Proceedings of the 1st IEEE International Conference on Cloud Computing in Emerging Markets (CCEM '12)*.

[21] Joel H. Schopp, Keir Fraser, and Martine J. Silbermann. 2006. Resizing Memory with Balloons and Hotplug. In *Proceedings of the 2006 Ottowa Linux Symposium (OLS '06)*.

[22] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*.

[23] Zhigang Wang, Xiaolin Wang, Fang Hou, and Yingwei Luo. 2016. Dynamic Memory Balancing for Virtualization. *ACM Transactions on Architecture and Code Optimization* 13, 1 (2016).

[24] Yi Wei, M. Brian Blake, and Iman Saleh. 2013. Adaptive Resource Management for Service Workflows in Cloud Environments. In *Proceedings of the 27th IEEE International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPSW '13)*.

[25] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacigümüş. 2011. Intelligent Management of Virtualized Resources for Database Systems in Cloud Environment. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE '11)*.

[26] Lenar Yazdanov and Christof Fetzer. 2013. VScaler: Autonomic Virtual Machine Scaling. In *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD '13)*.

[27] Qi Zhang, Ling Liu, Jiangchun Ren, Gong Su, and Arun Iyengar. 2016. iBalloon: Efficient VM Memory Balancing as a Service. In *Proceedings of the 23rd IEEE International Conference on Web Services (ICWS '16)*.