Application Level Ballooning for Efficient Server Consolidation

Tudor-Ioan Salomie Gustavo Alonso Timothy Roscoe
Systems Group, Computer Science Department
ETH Zurich, Switzerland
{tsalomie, alonso, troscoe}@inf.ethz.ch

Kevin Elphinstone
UNSW and NICTA, Australia
kevine@cse.unsw.edu.au

Abstract

Systems software like databases and language runtimes typically manage memory themselves to exploit application knowledge unavailable to the OS. Traditionally deployed on dedicated machines, they are designed to be statically configured with memory sufficient for peak load. In virtualization scenarios (cloud computing, server consolidation), however, static peak provisioning of RAM to applications dramatically reduces the efficiency and cost-saving benefits of virtualization. Unfortunately, existing memory "ballooning" techniques used to dynamically reallocate physical memory between VMs badly impact the performance of applications which manage their own memory. We address this problem by extending ballooning to applications (here, a database engine and Java runtime) so that memory can be efficiently and effectively moved between virtualized instances as the demands of each change over time. The results are significantly lower memory requirements to provide the same performance guarantees to a collocated set of VM running such applications, with minimal overhead or intrusive changes to application code.

1. Introduction

Virtualization in cloud computing and server consolidation enables applications previously deployed on dedicated machines to share a physical server, reducing resource consumption, energy, and space costs among other benefits. Statistically multiplexing such servers and ensuring application performance as load changes, however, requires careful coordination policies, and virtual machine monitor (VMM) mechanisms like live migration and memory ballooning are used to dynamically reallocate resources such as RAM in virtual machines (VMs) to meet performance goals.

Eurosys'13 April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04...$15.00

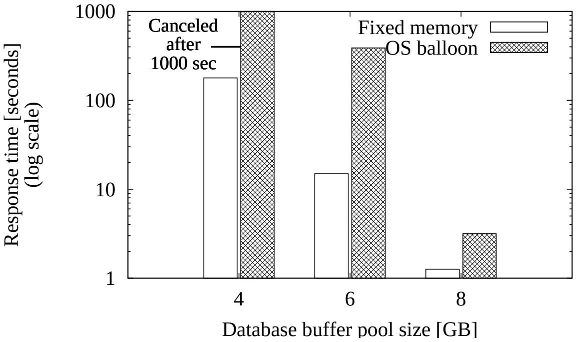


Figure 1. Effect of conventional ballooning on query performance (TPC-H, Q19)

These techniques work well when the OS manages application memory via paging. However, they do not work well when the application manages memory itself – the common case in server applications like databases and language runtimes. Here, efficient execution depends on the program having an accurate picture of available resident memory. Reallocating RAM on these systems using standard ballooning severely impacts performance, leading to thrashing and, in some cases, failure. A database, for example, allocates memory internally from a large, fixed size pool acquired at startup which it assumes is mostly resident. Standard ballooning transparently pages this pool to disk, causing the DB to choose the wrong query plans as data it thinks in main memory is actually not there. Figure 1 shows this effect when running a query from the TPC-H benchmark on MySQL. The figure shows the response time for the query with the database operating on 4, 6, and 8 GB of actual machine memory and when the memory has been ballooned down from 10 GB to 4, 6, and 8 GB using standard ballooning techniques [2]. That is, in each pair of experiments, the database has the same amount of machine memory. However, in the case of ballooning, the fact that some of the memory it thinks it has is actually not there has a catastrophic effect in performance. This effect is well known in industry and has already been mentioned by other researchers [6].

In this paper, we present *application-level ballooning* (ALB), a technique for reallocating RAM among a collec-

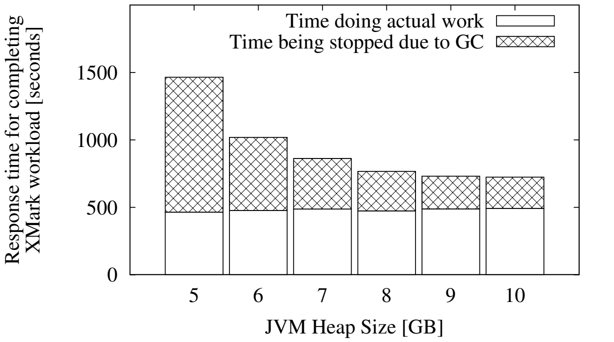


Figure 2. Response time vs. JVM heap size

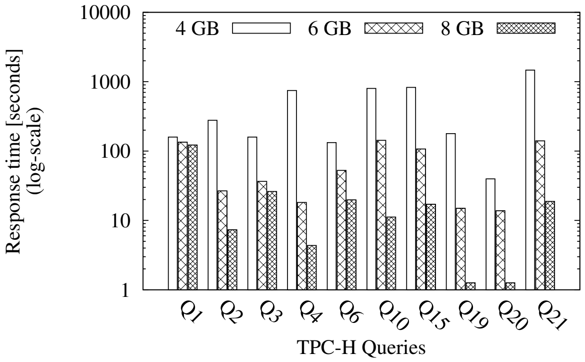


Figure 3. Response time vs. configured DBMS memory

tion of applications that manage their own memory running in collocated VMs. ALB re-allocates RAM without shutting applications down or reconfiguring them. It is fast, effective, requires minimal changes to applications, and crucially preserves the ability of an application to optimize performance based on having an accurate idea of the available physical RAM: with ALB, the query shown in Figure 1, has the same performance as when there is no ballooning involved.

ALB makes several contributions. We show in Section 2 how measured performance of two quite different applications (MySQL and the OpenJDK Java runtime) usefully correlates with available memory, motivating the ability to reallocate RAM to preserve SLAs while maximizing utilization. We then present in Section 3 the design of ALB, showing how it coordinates application, guest OS, and VMM. We extend existing ballooning mechanisms in the system software stack so that physical frames can be identified using virtual addresses. We have implemented ALB in Xen with Linux as a guest and MySQL and OpenJDK as applications.

Section 4 gives a more detailed evaluation of the ALB mechanism, showing that ALB can quickly and efficiently reallocate memory between applications. We show ALB's minimal runtime overhead versus applications, and confirm the conventional wisdom that guest OS paging alone typically results in orders of magnitude of slowdown and is not an acceptable solution, Finally, we demonstrate that ALB can trade off performance with memory usage in both MySQL and OpenJDK, and dynamically optimize end-to-end performance of a two-tier application by moving memory between tiers.

2. Motivation

We target applications that manage their own memory, are statically configured with a fixed quantity of memory, and optimize their use of this pool to maximize performance. Most enterprise applications fall into this category. Here we consider databases, which extensively cache data and results to avoid I/O, and language runtimes, which exploit extra memory to reduce the frequency of expensive garbage collections.

2.1 Memory allocation and performance

Ballooning is useful if varying the actual memory allocated to an application changes its performance in a stable way. That such is the case for sever applications like databases and language run times is easy to demonstrate. Figure 2 shows the response time for the XMark XML benchmark [24] as we vary the configured JVM heap size: additional memory dramatically reduces the overhead of garbage collection, resulting in significant overall speedup although the amount of actual effective work done remains the same.

Figure 3 shows how the memory size of a database running the TPC-H benchmark [27] affects query response times (note the log scale). Increasing available memory from 4GB to 8GB reduces the response time of most (but not all) TPC-H queries, sometimes by two orders of magnitude, mostly due to reduced I/O. Databases manage their own pool of memory because they have knowledge of access patterns and relative page utility, knowledge not available to the OS.

Such a significant influence leads to two problems with statically provisioning RAM to VMs, even when informed by load profiles [25]. The first is the need to provision for peak load, leading to inefficiencies as it is expensive to leave memory idle, the space-time tradeoff depends heavily on workload, and less applications can be collocated in a physical machine. The second is fragmentation when starting, stopping, or migrating VMs, which creates a packing problem across physical machines.

2.2 Reconfiguring memory allocation

Effective application consolidation though virtualization requires the ability to change the size of machine memory available to an application. We discuss four possible solutions to this problem.

Restart: We can stop the application and restart it with a larger memory pool backed by the VMM. For example, a DB would stop accepting queries, wait for existing queries to complete, shut down, and restart with a larger memory pool. It requires no changes to the DB, OS, or VMM, but entails significant downtime and hence reduced availability. In the case of JVMs application restart might be optimized [3, 12], but restart is still a poor fit for high availability scenarios.

Paging: The OS can demand-page application memory to disk, perhaps in response to VMM memory ballooning. The problem is that server applications like databases pin memory to RAM precisely to avoid OS level paging. Like restart, paging can reduce an application's RAM requirements without code changes. However, paging undermines the optimizations performed by a database or garbage collector: it becomes impossible to intelligently manage database buffers, for example. In the worst case, *double paging* occurs: the application touches a page in its memory in order to evict it and write it to disk, unaware that that page has already been paged out. The page is needlessly read in from disk (evicting another VM page in the process), only to be written back and discarded.

Rewrite: Applications can be rewritten to rely on the OS for resource management [7], no longer assuming a fixed memory pool. Long-term, we believe this will happen: it is increasingly commercially important for a database to share a machine with other applications, for example, and designing an DB in this way is a promising research topic. Short-term, however, the engineering cost is prohibitive. Databases and language runtimes represent decades of careful engineering, with many feature interdependencies tied to many critical legacy applications, and a radically redesigned storage manager or garbage collector for modern hardware (physical and virtual) will take years to mature.

Conventional ballooning: A guest OS itself shares the characteristics of our example applications, and VMM designers rejected restart, paging, and rewrite as ways to vary the machine memory allocated to a guest OS, in favor of *memory ballooning* [29]: each guest kernel incorporates a *balloon driver*, typically a kernel module, which communicates with the VMM via a private channel. To the guest OS, the balloon driver appears to allocate and deallocate pinned pages of physical memory[1]. To reduce the memory used by a VM, the driver "inflates the balloon", acquiring physical memory from the kernel's allocator much as a device driver would allocate pinned DMA buffers. These pages are now considered by the guest OS to be private to the balloon driver, which notifies the VMM of the pages it has acquired. The VMM converts the physical addresses to machine addresses, and reclaims these pages for use by other VMs. Conversely, "deflating the balloon" returns pages from the VMM to the balloon driver, which then "frees" them in the guest OS. Compared with the VMM transparently paging the guest, ballooning dramatically improves performance for applications whose memory is managed by the OS. Inflating the balloon increases memory pressure inside the guest, making to page its own memory to virtual disk, thereby making a more informed page replacement choice than the VMM can. As shown above, conventional ballooning does not work for applications that manage their own memory.

2.3 Requirements for ALB

ALB enables ballooning for applications that manage their own memory by inserting a balloon module which allocates memory from the application's pool and returns it to the OS via a system call, and extending the OS to return pages to the application via the module. The *virtual* memory used by the application can therefore be dynamically varied between a minimal level and the full configured size.

However, while attractive, ALB's effectiveness depends on some key assumptions. We discuss them below, and justify them with the aid of results from Section 4. First, available memory at runtime must *correlate with performance* for most workloads. Figures 2 and 3 show two clear examples of this happening. Furthermore, for most workloads we see a clear "knee" in the curve: a threshold of memory size above which the system always performs well. Second, ALB *must not violate application assumptions* about available memory. A database query planner might base optimization decisions on the statically configured memory size, whereas in reality much of that space has been ballooned out, causing worse performance than static configuration with less memory. This was shown in Figure 1. That is, ballooning must not result in the application becoming delusional about where data resides. Third, the *engineering effort* to modify application, guest, and VMM must not outweigh the benefits. We quantify the (small) code changes needed in Section 4.2. Fourth, applications should *respond quickly* to changes in memory size due to ballooning. Databases heavily caching data require time to "warm up" newly available memory for buffers, whereas a JVM can immediately allocate objects from new heap space. In Section 4.7 we evaluate our applications' responsiveness to changes in memory allocation.

Finally, ALB as currently designed is not suitable for all applications. Some, such as PostgreSQL, delegate their memory management to the OS (via the file system buffer cache), rendering ALB redundant. Others, such as the Tomcat application server, implement another layer of memory management above the JVM, requiring an additional layer of ballooning.

3. System design

ALB enables the reallocation of machine memory (RAM) between memory managers of different applications (databases and JVMs) residing in different virtual machines as depicted in 4, while preserving the performance of each as a function of its currently available memory.

We focus in this paper on the *mechanism* for reallocation between applications and mostly control ballooning manually in our experiments, though we show a simple controller

[1] Henceforth, we use the customary terminology: *physical memory* refers to a VM's memory that the guest OS manages, and *machine memory* denotes real RAM managed by the VMM.

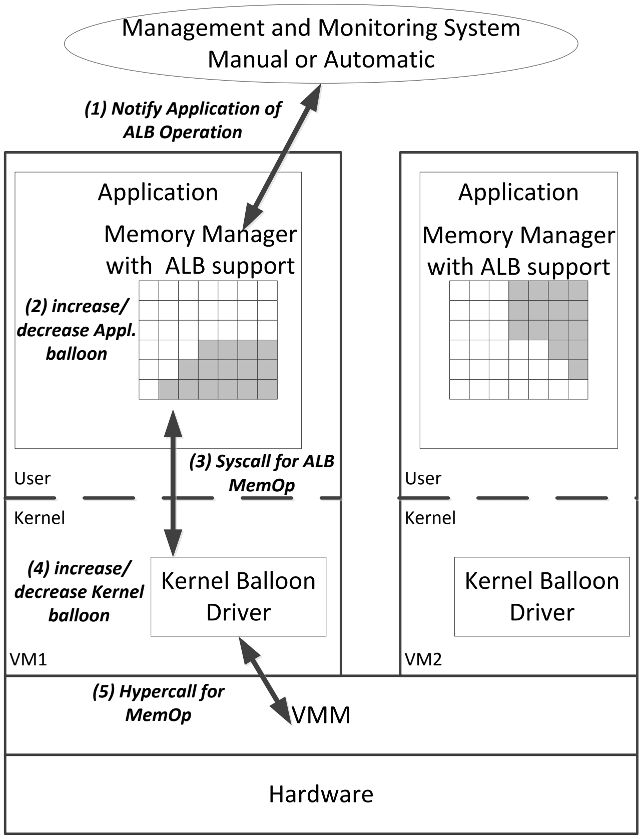

Management and Monitoring System
Manual or Automatic
(1) Notify Application of ALB Operation
Application
Memory Manager with ALB support
(2) increase/ decrease Appl. balloon
Application
Memory Manager with ALB support
(3) Syscall for ALB MemOp
User
Kernel
User
Kernel
(4) increase/ decrease Kernel balloon
Kernel Balloon Driver
Kernel Balloon Driver
VM1
VM2
(5) Hypercall for MemOp
VMM
Hardware

Figure 4. ALB System architecture & Call stack

at work in Section 4.6. We leave the *policy* issue of how to autonomically reallocate memory for future work,

We implemented ALB for MySQL and OpenJDK, using a Linux guest OS and Xen. We exploit the existing balloon functionality in Xen, and extend the applications to support ballooning pages to and from MySQL's buffer pool and OpenJDK's heap. We modified the existing balloon driver in Linux as it was only designed to balloon pages from the kernel free list. Furthermore, ALB refers to ballooned pages by virtual addresses, requiring additional address translation.

The balloon module in ALB-aware applications interfaces with both the kernel and a management system. The former occurs via a system call which frees or reclaims memory for the application's balloon module. While we could have overloaded the **mmap** and **munmap** system calls to achieve this, we chose to add a new, orthogonal system call to allow flexibility in the OS, and to support applications which do not allocate memory at startup using **mmap**.

The management system determines policy, and conveys a target balloon size to the ALB module via an RPC call to a local socket. Applications which already have user-facing interfaces can also support other interfaces – for example, we also added SQL ballooning extensions to MySQL for testing.

Figure 4 shows ALB in operation. The management system controls the ballooning process by changing the balloon target for the application (1). The application allocates pages (2) and notifies the OS via a system call (3). The modified guest OS balloon driver processes the requests (4) and makes the memory changes visible to Xen with a hypercall (5).

We now discuss in more detail the modifications required to MySQL, OpenJDK, and Xen/Linux to make ALB work.

3.1 The MySQL ALB module

Our balloon module for MySQL is built into the InnoDB storage back-end. InnoDB manages its own memory, creating a fixed-size buffer pool at startup divided into 16kB InnoDB "pages" and is used for both the lock table and data cache. Most pages are cache, managed with an LRU list.

The balloon module uses the internal `mem_heap_create_in_buffer` and `mem_heap_free` calls to acquire and release InnoDB pages. Once acquired, a page will not be used by the pool for caching data and is effectively removed from the available memory until freed.

To inflate the balloon, ALB acquires memory from the pool and adds it to a list of ballooned InnoDB pages; there is one such list per database instance. A list of the aligned virtual address ranges for pages is passed via the new system call to Linux for VM ballooning. To deflate the balloon, ALB traverses the list of ballooned pages to be returned to the LRU free list and notifies the kernel of the number of pages required. The pages are then faulted in by zeroing out the memory at the corresponding virtual addresses.

There are some subtle implementation issues. 16kB InnoDB pages consist of four 4kB virtual pages, and InnoDB stores accounting metadata in the first of these when the InnoDB page is on the free list. A naïve ballooning module would therefore only be able to return 75% of the virtual pages to the OS (first 4kB virtual page of the 16kB InnoDB page cannot be ballooned). Our implementation of the ballooning module in MySQL/InnoDB includes an optimization that copies the metadata (less than 300 bytes) from the first 4KB virtual page into a new, pre-allocated area of memory in InnoDB. This requires about 300MB extra memory to balloon 16GB, but the net result is increasing utilization to 98%.

3.2 The OpenJDK ALB module

The ALB module for OpenJDK is more involved than in MySQL. While InnoDB uses most of the memory as cache, the JVM passes most of its memory to the Java heap controlled by the garbage collector. For the JVM we therefore focus on ballooning in and out of the heap space, and modify the *Parallel Scavenge Garbage Collector* (PSGC) that ships with OpenJDK7.

PSGC is a generational collector; Figure 5 shows the structure of the heap. The *permanent generation* holds class metadata. The *young generation* holds recently-created objects, and those that have survived a few garbage collections. Objects that survive longer are moved to the *old generation*. The *young generation* is also split into *eden* and *survivor*

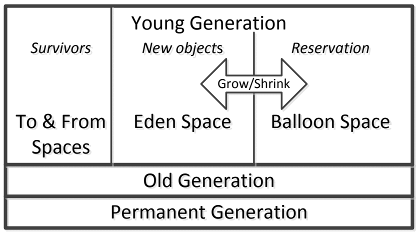


Figure 5. Structure of OpenJDK's parallel scavenge heap

spaces. Spaces in PSGC are bounded by start and end addresses, and all allocated memory sits between the *start* and *top address*, which is incremented as objects are allocated. Collection compacts space between the start and top addresses, removing holes and moving objects to other spaces.

We implemented JVM ballooning as a new, *balloon* space in the *young generation* which can grow and increase the pressure on the *eden* space when necessary, or contract and reduce the pressure. Our current implementation only balloons the *eden* space and does not support PSGC's adaptive size policy, though the design does not prevent us supporting more spaces/generations and their adaptive resizing.

In order to resize the spaces composing the heap, we need to compact them and prevent any other operations during ballooning. For this reason, the ballooning operation is performed at the same time as a full garbage collection. Before returning from a full collection, we perform all outstanding ballooning operations. This means that the cost for ballooning operations in the JVM is influenced by the time needed to perform a garbage collection, as detailed in Section 4.7. A tighter coupling of the GC implementation with ALB would reduce much of this overhead.

Besides the balloon reservation mechanism, a new JVM thread monitors incoming requests for balloon operations and invokes the garbage collector to perform a compaction followed by the resizing of the *balloon* space.

We considered an alternative, garbage-collector-agnostic solution which simply creates special Java objects to inflate the balloon, as in a recent patent from VMware [15]. Unfortunately, this approach has problems. First, translating from an object reference to an OS virtual address, without modifying the JVM, requires calling into native code via JNI (since the object address is not available within Java).

Second, most GCs move objects among and within heap spaces, requiring re-translation of the object address after each collection phase. OpenJDK does offer a runtime callback for garbage collection completion, but it would still result in considerable runtime overhead in remapping pages. Alternatively, objects used for memory reservation could be pinned in memory using JNI, though the implications of this vary between GC implementations.

Consequently, it is not possible to be fully independent of the garbage collector. The design we adopted does modify the PSGC, but it does not depend on JNI or runtime callbacks, it does not require address translations, and it is not impacted by regular GC operations, thereby reducing the code complexity of the balloon mechanism.

3.3 Changes to the Linux Kernel

To support ALB, we modified a Xen-patched Linux kernel so that it can be notified of ALB operations via a new system call, which required relatively little code. The new system call takes as parameters (i) the ALB operation, (ii) the start of the virtual memory address and (iii) the size of the memory on which to operate. A pseudo-device or `/proc` file system entry would work as alternatives to our system call approach.

The ALB operation either inflates or deflates the balloon. The system call handles an inflate request by ensuring the candidate virtual pages refer to resident physical pages by faulting them in. Once resident, the physical page numbers are obtained, and a request to increase the kernel balloon is enqueued with Xen's balloon driver. The kernel handles a deflate request by taking the number of OS pages required and enqueueing them with the kernel balloon driver. On return, Linux's free list will have expanded to support subsequent use of the deflated application pages.

3.4 Changes to the Xen Balloon Driver

Xen's balloon driver processes operations via a kernel work queue. For ALB operations, work items are enqueued as a result of executing the ballooning system call.

A `balloon_process` function dequeues work items and either increases or decreases the kernel balloon. The same function handles both traditional ballooning requests and ALB requests. The main difference lies in the source of the physical pages that are ballooned. In one case these pages are allocated from or freed to the Linux kernel, while in the case of ALB the physical pages are taken from and returned to the corresponding application's virtual memory area.

Both InnoDB and OpenJDK allocate their memory pool through an `mmap` operation. The requested memory is anonymous and private (meaning that it can be swapped, is not backed up by any file descriptor, and is zero-filled). Virtual addresses from the application might or might not have been faulted in the guest OS. For ballooning out a page, we have to ensure that the physical page backing a virtual address is faulted in: we use the `get_user_pages` function [14] for this. Faulting in the pages will add entries in the guest OS's page table. Based on the `mmap` request, the physical pages obtained from the application are anonymous, private, with a map count of 1, and with a valid mapping pointer. Also they are present in both the page table of the guest OS and the kernel's LRU list.

In contrast, the pages that are used by Xen for traditional ballooning are newly allocated in the kernel through a call to `alloc_page` [14]. Besides having another state & flags, they are not in the kernel's LRU list.

Xen's *hypercalls* for memory operations require that the pages moved from the guest OS into the VMM be in the same state as those obtained from a call to alloc_page. This means that pages backing up an application's mmap-ed memory require extrication for use by the balloon driver.

When increasing the balloon, for each page that backs an application virtual address, we wait for the LRU to drain (i.e., for the page to be faulted in), clean its contents, move it from the LRU list to the ballooned list, and clear its mappings and anonymous state. Once the page is "sanitized" the page table is updated so that the virtual address no longer maps to this page. Next, the virtualization mapping between page-frame-numbers (PFNs) and machine-frame-numbers (MFNs) is removed. Finally, a hypercall notifies the VMM that the page is free and the guest OS has released control of that memory.

When decreasing the balloon, a buffer that will be populated by the VMM is initialized with pages removed from the list of ballooned-out pages. Once reclaimed through a hypercall, these pages are mapped in the page table at the correct application virtual addresses (these can then be re-enabled in the buffer pool/ heap). The map count, page count and anonymous mappings are re-established, as well as the virtual memory page protections. Once this is done, the page is linked back into the kernel's LRU list.

4. Experimental evaluation

We now present our evaluation of ALB implemented as described in Section 3. The aim of our evaluation is fivefold:

First: We compare the performance of a conventionally sized (i.e. statically provisioned) application with that of the same application sized by shrinking using ALB.

Second: We show the performance characteristics over time of a system with two tenant applications, where ALB is used to dynamically reallocate memory between them.

Third: We examine a simple two-tier architecture, which relies on MySQL for the data tier and on the JVM for the business tier. In this setup, we move memory during runtime between the data tier and the business tier, in order to reduce the overall latency of requests to the system.

Fourth: We show a setup with four collocated database that can react to workload changes, mitigating the performance degradation caused by spikes in the workload through ALB.

Fifth: We measure the performance of balloon inflation and deflation in both the database and the JVM.

4.1 Experimental setup

We used a 2 x Quad Core AMD Opteron 2376 2.3GHz Processors, with 16GB of DDR2 RAM, and two 7200RPM 1TB SATAII drives server for the overhead, in-flight and end-to-end experiments (Sections 4.3, 4.4, 4.5) and a 64 Core AMD Opteron 6276, 256GB of DDR3 RAM and 4 x OCZ Vertex 4, 256GB SATAIII SSD drives for the database collocation experiment (Section 4.6). We used Xen 4.1 [31] as the VMM, with 64bit Ubuntu 10.04 (Linux kernel version 2.6.32) with Xen patches running in Domain0 as the management OS. The guest OSes were paravirtualized instances of 64-bit CentOS 6.2 (Linux kernel 2.6.32), with the VMs configured with 4 virtual cores each. The database engine was 64-bit MySQL 5.5 with the InnoDB storage engine. The JVM was a 64-bit OpenJDK7. Both MySQL and OpenJDK were compiled with gcc 4.4.3. Xen and Domain0 boot live images via PXE Boot and run from an in-memory file-system, removing disk interference caused by Xen itself from our experiments.

On this platform we evaluated the characteristics of database ALB using queries from the TPC-H benchmark (using a dataset with a scale factor of 5, corresponding to approx. 5GB of raw data) and those of JVM ALB using the XMark benchmark (over an XML dataset of 1GB) relying on the Nux toolkit [19] with Saxon 8 [23].

4.2 Engineering cost

The cost of implementing ALB is modest requiring 870 lines new lines to the Linux guest kernel, 229 lines to MySQL, and 427 lines to OpenJDK. This suggests that the changes required to application memory managers to support ALB are small, and that most of the complexity is abstracted in the kernel including the existing kernel-level balloon driver.

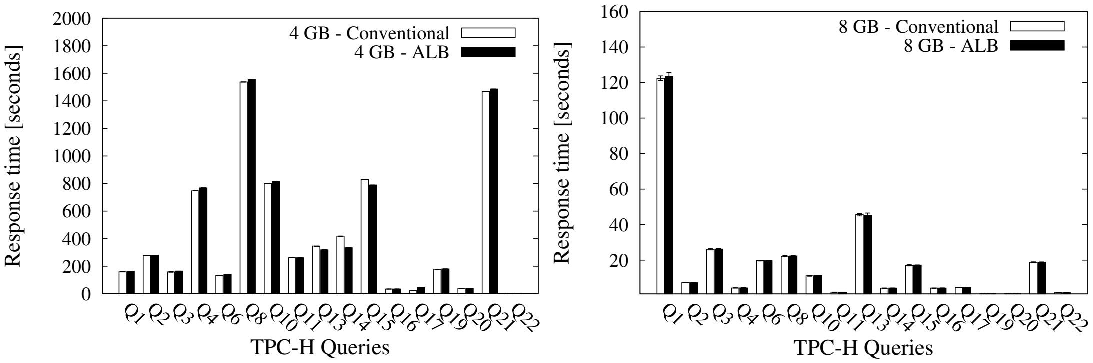
4.3 Overhead of ballooning

In order to understand the overhead of ALB, we investigate the performance of MySQL and OpenJDK under the following two scenarios:

Conventional: MySQL and OpenJDK run without any changes in a guest OS. The guest runs a paravirtualized Linux kernel, with the default Xen patches. While the guest OS binaries run from a RAMdisk, the MySQL database files and the XMark input XML file are stored directly on a dedicated real disk. The amount of memory allocated to the InnoDB buffer pool is varied between 4 and 10GB by directly changing MySQL's configuration file. The amount of heap space allocated to the JVM is varied between 5 and 10GB. These provide the performance baselines for statically provisioned MySQL and OpenJDK instances.

Ballooned: MySQL and OpenJDK run with their individual ALB modifications in place, within a paravirtualized Linux guest as above. Additionally, the guest OS now contains our modifications to Linux and the balloon driver to support ALB. Again, the guest OS binaries are on a RAM-disk, and the data files are stored directly on a dedicated real disk. However, in this case the amount of memory available for the InnoDB buffer pool and the JVM's Heap are directly controlled via ALB. For each of the memory sizes above, we start MySQL and OpenJDK statically provisioned with the maximum memory size of 10G, and then balloon out pages in order to reach the desired size. We then measure the performance with the reduced amount of memory.

Due to the large dataset size (TPC-H with a scale factor of 5) and our storage setup, run time for some of the TPC-H



(a) TPC-H Benchmark, SF5, 4GB Buffer Pool Size

(b) TPC-H Benchmark, SF5, 8GB Buffer Pool Size

Figure 6. Ballooning overhead in MySQL: Conventional vs. Ballooned query response times

queries was prohibitively long. We removed from our experiments queries that failed to complete within 30 minutes. Consequently, we present results for 17 queries out of 22. These are queries 1–4, 6, 8, 10, 11, 13–17, and 19–22. For the XMark queries, we omit running queries 8 through 12. These 5 queries are large "join" queries that failed to complete within 2 hours (on the 1GB input XML data file). Each XMark query is run 10 times in all the experiments, and we report the total time.

While the omitted queries dramatically extend the duration of each experiment, they behave similarly and so do not affect the results significantly.

4.3.1 MySQL overhead

Figure 6 shows the average response time of each query, in the two scenarios (Conventional and Ballooned), for two memory configurations: 4GB and 8GB. Figure 6(a) shows that the ballooned configuration performs almost identically to the conventional configuration for an I/O intensive setup, where most of the data can not be cached in the buffer pool. Similarly, Figure 6(b) shows that there is almost no overhead between the ballooned and conventional configurations for CPU intensive setups where most of the data is cached in the buffer pool. From Figure 6 we conclude that there are no substantial differences between the two configurations we investigated. The differences we do see are caused by randomness in the query parameters and non-deterministic variation in the system state (e.g., MySQL's random probing for cardinality estimation).

4.3.2 OpenJDK overhead

Figure 7 shows total runtime for the sequence of 15 XQueries in the XMark benchmark. Each bar depicts total run time, divided into time the JVM did useful work in answering the XQueries and the time the JVM spent on garbage collection (due to the small heap). In all experiments the workload stayed CPU bound. We draw two conclusions.

First, increasing the JVM heap reduces the time overhead of garbage collection, thus speeding up query response time.

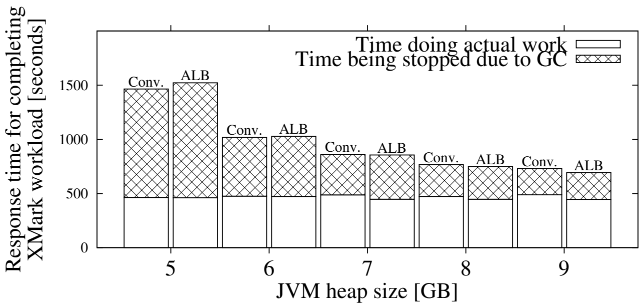


Figure 7. Ballooning overhead in OpenJDK: Conventional vs. Ballooned runtime

Second, for all of the five memory configurations ranging from 5 to 10GB for the heap size, the performance of the ballooned configuration ("ALB") closely resembles that of the conventional configuration ("Conv.").

4.4 In-flight memory resizing

We now examine the performance characteristics of using ALB to reallocate memory between two virtual machines. In one experiment we resize MySQL/InnoDB, in the other we resize OpenJDK. Two clients issue the application specific workload to stress the corresponding application.

4.4.1 Resizing MySQL's buffer pools

For running the memory reallocation between two MySQL instances, we first bring up a database (VM1-DB) in one VM configured to a total of 9GB of RAM, 8 of which reserved for the InnoDB buffer pool. We then reduce this allocation by 4GB using ballooning, allowing us to bring up another database (VM2-DB) in the other VM, also configured to a total of 9GB with 8 reserved for InnoDB.

The workload used for this experiment consisted of the TPC-H queries 2, 3, 6, 11, 16, 17, 19, 20, and 22, selected because they have lower response times and allow us to present the throughput of the system as a function of time without having to aggregate over long periods.

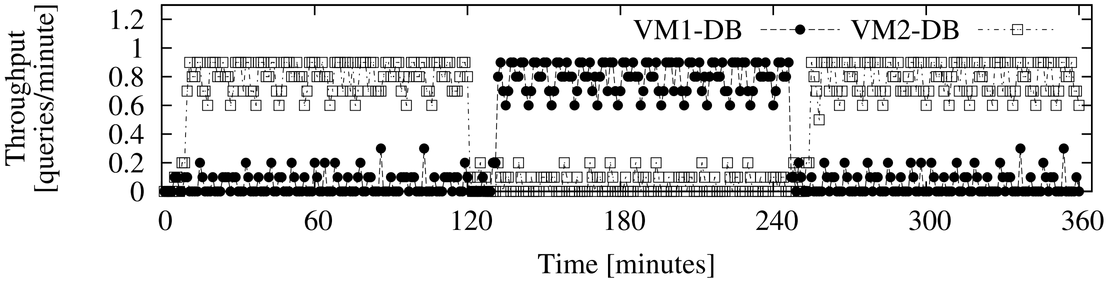


Figure 8. TPC-H Benchmark throughput evolution as we reallocate memory (4GB) between two DBs across VMs

During a 6 hour test, we changed the buffer pool size of the InnoDB storage engines every 2 hours by reallocating 4GB back and forth between the two databases using ALB. After 120 minutes, we reduce the buffer pool of VM2-DB by 4GB and increase that of VM1-DB by the same amount. After 240 minutes, we reverse the process.

Figure 8 shows the throughput of the two systems as a function of time. The throughput is reported every 2 minutes. When the in-flight resizing operations occur, the throughput of the two systems change. At minute 120 we observe a rapid drop in throughput for VM2-DB (a large number of ballooned-out InnoDB pages will lead to more disk access for answering queries). At the same time, a steady increase in the throughput of VM2-DB is observed.

The gradual increase in throughput for VM2-DB is caused by the database warm-up period. It takes time for the new pages that were ballooned-in to the buffer pool to be actively used. Here we are performing a somewhat extreme experiment in which we move 4GB of memory, meaning that 50% of the buffer pool needs to be warmed up to reach steady-state performance. The same behavior can be observed for the in-flight resizing performed at minute 240.

Two features of the results stand out: the oscillating throughput and the apparently long delay before the database can fully exploit newly ballooned-in pages. The oscillating throughput is a consequence of the nature of the workload. The queries used in the workload have very different response times. The dataset is too large for main memory, forcing InnoDB to frequently access the disk. This behavior prevents us from running a large number of clients in order to observe a more stable throughput, as disk contention from multiple clients would make this experiment prohibitively slow. Also, the wide variation in response time across different queries (see Figure 3) leads us to aggregate throughput figures over bins of 2 minutes. The apparently long warm-up times (minutes 120-to-130 for VM1-DB and 240-to-250 for VM2-DB) are caused by the cache warm up time. The queries running at the point when memory is ballooned-in determine how fast the buffer pool is populated. For reference, the same warmup phenomenon is seen in the figure at time 0, when the database starts with an empty data cache.

4.4.2 Resizing the JVM heap

As in the previous experiment we start the VMs sequentially. The first VM, configured to a total of 8GB, out of which we allocate 7GB to VM1-JVM. These 7GB are split into the different parts of the heap: a fixed 3GB old generation space, and a fixed 4GB for the young generation, including a 256MB to-space and a 256MB from-space, leaving 3.5GB for the eden space. We continue expanding the JVM balloon over 2GB from the eden space. With the newly released memory, we can start the second VM, giving it the same total memory of 8GB, out of which we allocate 7GB to VM2-JVM (again with 3GB for the old generation, 3.5GB for eden space and 256MB for to-space and 256MB for from-space).

Similar to the MySQL in-flight-resizing experiment, we run a 6 hour test, during which we reallocate 2GB of memory between the heaps (eden-space) of the two JVMs every 2 hours. The results are presented in Figure 9. At minutes 120 and 240 we notice how the throughputs of the two JVMs change. When the eden spaces increase from 1.5GB to 3.5GB (minute 120 for VM1-JVM and minute 240 for VM2-JVM), we see an instant increase in throughput. Comparing to the case of buffer pool of MySQL, the JVM can instantly make use of the extra memory without needing to warm up a cache.

4.5 An end-to-end example

ALB is not restricted to reallocating memory between the same type of application. In this experiment we move memory between MySQL and OpenJDK. The motivation is a simple two tier architecture, in which the data tier is implemented by a MySQL database running in a VM, the business tier is implemented by a Java application that processes XQueries, running in another VM. The communication with clients is done via a minimal socket-based interface for receiving queries and sending results back.

The synthetic workload that drives this benchmark requires that for each request a set of database queries needs to be run in the data-tier. Based on their results, a series of XML document processing operations will be performed in the business tier, and the results sent back to the client.

In the synthetic workload, for each client request, we perform a series of database queries (we ran the TPC-H

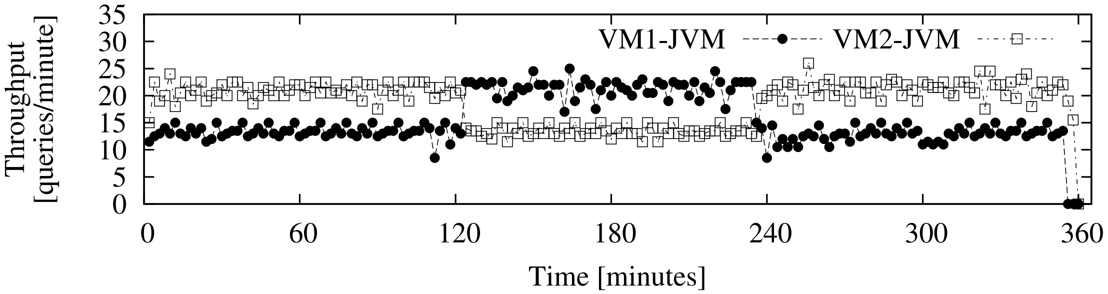


Figure 9. XMark Benchmark throughput evolution as we reallocate memory (4GB) between two JVMs across VMs

queries 2, 3, 6, 11, 16, 17, 19, 20 and 22) on a TPC-H SF5 dataset, followed by a series of XMark queries (1-7 and 13-20) on a 1GB input XML file. The response sent back to the client is the total runtime for each individual query.

The experiment starts by creating a VM with a total of 9GB of RAM, in which we start VM1-DB, a MySQL instance with a buffer pool of 7.5GB. We then balloon 1.5GB out from the database, reducing its actual buffer pool to 6GB. Next we start a second VM with a total of 7GB of RAM, in which we create a VM2-JVM with a fixed heap of 6GB (3GB old generation, 2.5GB eden-space and the remaining 0.5GB equally split among the to/from spaces). The initial configuration *Static* in Table 1, runs the workload without performing any memory movement between VM1-DB and VM2-JVM. VM1-DB runs with an actual buffer pool of 6GB and VM2-JVM runs with a fixed heap of 6GB. The second configuration *Dynamic* in Table 1 starts in the same configuration as the *Static* one, but reallocates 1.5GB from VM2-JVM to VM1-DB before performing any database queries, and then moves the 1.5GB back to VM2-JVM before performing any XML processing.

Config	DB Queries (sec)	XML Queries (sec)	Balloon Ops (sec)	Total (sec)
Static	2077.25	455.97	0	2533.22
Dynamic	164.57	450.37	21.53	638.00

Table 1. Runtimes for whole system: Static vs. Dynamic configs.

Table 1 presents the runtime break down for the two system configurations. For the *Static* configuration we do not spend any time on doing ballooning operations. In the *Dynamic* configuration we see that at the cost of 2 ballooning operations (each taking approx. 21 sec.) an overall improvement of 4x is gained. This approach of moving memory between the data and business layer only makes sense if the gain in query response latency covers the cost of the ballooning operations. Our current implementation of ALB is not suitable for fast, interactive systems, where the overall latencies need to fall under 5 sec.

4.6 Collocating database servers

We now present a large scale experiment that highlights the benefits of ALB in the context of collocated database servers.

For the experiment we used a 64 core, 256GB RAM AMD machine, running Xen: Domain0 on 16 cores with 2GB of RAM and 4 VMs, each having 12 cores 36GB of RAM. Within each VM a MySQL / InnoDB database operates on a TPC-E [26] dataset of 105GB (TPC-E scale factor of 5000 customers). The TPC-E dataset consists of brokerage transactions corresponding to different customers and the workload comprises mostly fast running read-transactions. All 4 databases have an initial InnoDB buffer pool of 8GB which can be increased up to 24GB through ALB.

The experiment consists of running a base workload (10 client threads, targeting 1000 of the 5000 TPC-E customers in the dataset) for a duration of 5h. An extra workload (20 client threads, targeting any of the 5000 customers in the dataset) increases the load on a database server. When a server receives extra load, the ALB will be triggered to increase the size of the buffer pool on the pressured database. Increasing the buffer pool mitigates the drop in performance due to the extra load. Once the extra load stops, we shrink the buffer pool.

Figure 10 shows the throughput of the 4 databases, each under a constant base workload and spurious extra workloads. At the 30 minute mark the first database starts receiving the extra workload. Due to pollution of the buffer pool (more data touched by the extra clients), there is an abrupt drop in throughput (top line in Figure 10). A monitoring system detects the extra load and the drop in throughput and reacts by increasing the buffer pool of DB1 from 8GB to 16GB [2]. Within 5 minutes, the throughput of DB1 recovers. After a total run of 40 minutes, the extra workload stops. When the extra workload stops (minute 70), the performance drop due to the reduced offered load and because the buffer pool still holding pages that are irrelevant for the base workload. Within 2 minutes, the throughput stabilizes again and the monitoring system initiates the shrinking of the buffer

[2] Implementing a monitoring system that triggers the buffer pool increase as well as determining the actual correlation between offered load and resource requirements are not in the scope of our work, but have been extensively studied [6, 25]

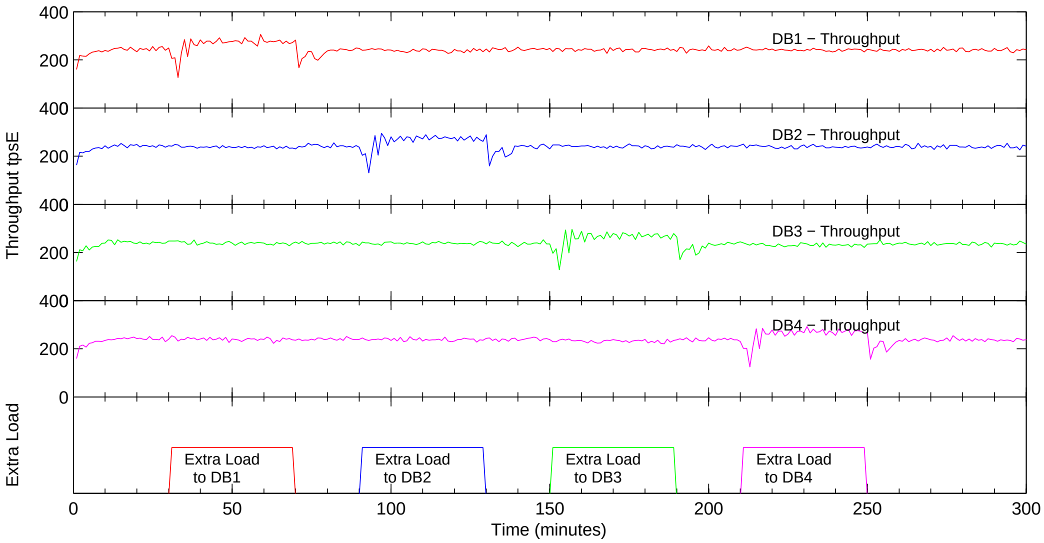


Figure 10. Collocating database servers: reacting to changes in workload

size of DB1 back to 8GB (minute 75). The second drop in throughput is caused by the shrinking of the buffer pool. Finally, at minute 80 minute the throughput of DB1 is stabilized. Extra workload events happen at minutes 90, 150 and 210 for DB2, DB3 and DB4 respectively. The behavior is the same as in the case of DB1.

This shows that ALB can be used in real scenarios of database collocation in virtualized environments and how spikes in load can be handled on the fly, without stopping the database, just by increasing the amount of cached data.

The reaction time to increasing the InnoDB buffer pool is determined by two factors: query diversity and the I/O subsystem. A high number of concurrent queries will lead to a fast cache population. Higher throughput from the I/O subsystem also speeds up cache population. In the current experiment we found that the time required to warm up the pages in the InnoDB buffer pool is bound by the I/O subsystem. Doubling the number of clients, the 90 percentile response time increases for some transactions by as much as 100% with an average of 62%. With 10 clients, the 90 percentile of response times of the transactions was in the range of [30-350]msec while with 20 clients it was [30-580]msec. This means that more load might decrease the cache warm up time, but would increase latency. Investigating the I/O subsystem, we saw that disks with fast seek times improve disk throughput (the TPC-E workload does many random data accesses [4]). For this collocation experiment, each database was stored on a OCZ Vortex 4, 256GB SSD drive. Using these drives (instead of traditional disks), the bottleneck shifted from the disk drives to Xen's VMM, where all I/O interrupts for the VMs were handled only on one core (having 100% utilization). Despite trying to balance interrupts among the VMM's 16 cores – we did not succeed. We speculate though that balancing the interrupts among more cores would further reduce the time required to warm up the database's buffer pools when increased.

4.7 ALB performance

We evaluate the performance of ALB operations by measuring the time it takes to increase or decrease the balloon, at application level, by a certain number of pages. We recorded the ALB operation response times (in Table 2), both for MySQL and for OpenJDK.

InnoDB Pages (16K)	Grow(sec)	Shrink(sec)
32768 Pages (0.5 GB)	3.99	0.77
65536 Pages (1 GB)	7.57	1.51
131072 Pages (2 GB)	16.07	3.02
JVM Pages (4K)	**Grow(sec)**	**Shrink(sec)**
131072 Pages (0.5 GB)	11.28 (2.14)	9.48 (0.41)
262144 Pages (1 GB)	16.34 (4.27)	9.96 (0.82)
524288 Pages (2 GB)	18.92 (7.68)	10.91 (1.65)

Table 2. ALB Grow/Shrink operations: Response times

For the case of InnoDB, we observe that there is a linear increase in the duration of both ALB operations with the ballooned size. The latency of the operations includes the application-level ballooning time and the kernel-level ballooning time. For InnoDB pages, the application level ballooning time is dominated by the `memcpy` operation of the InnoDB page metadata. This cost is symmetrical for the grow and shrink operations. The large 5x difference between growing and shrinking the balloon comes from the operations that we perform in the kernel. Compared to traditional ballooning, ALB operates on pages originating from `mmap`ed memory. Before we can place a page into the balloon, we need to remove it from the page's zone LRU. Unfortunately this operation can only be done under a kernel zone lock. The cost of acquiring this lock for pages ballooned out makes the grow operation more expensive and limits the achievable throughput for ballooning operations. We expect that improvements to the kernel, like the adoption of lock free data structures [5], would reduce the time needed to perform ALB balloon-increase operations.

For the case of the JVM, two numbers are presented for the grow and shrink operation. The first number is the total time it takes to perform the ballooning operation (both application and kernel part). The number in parentheses gives the total time spent in the kernel for completing the ballooning operation. For the kernel time, the same remarks apply as in the case of InnoDB. The large amount of time spent in the JVM for completing the ballooning operation is a consequence of the full garbage collection operation that we perform before each ballooning operation in the JVM. Section 6 looks into ways to further improve this.

5. Implementing ALB

This section conveys our experience in implementing the ALB modules for OpenJDK and MySQL/InnoDB as a set of engineering guidelines that can be reused in adding ALB modules to other applications.

ALB is a mechanism suited for applications that take memory management under their own control, bypassing the OS. In implementing an ALB module the developer needs to answer a set of questions:

1. What does the application use the memory for?
2. What data structure describes the free/used memory?
3. What are the policies for allocating and freeing memory?
4. How can the data structure and the policy be adapted to support memory reservation?

For most server class applications that do their own memory management, we have observed that they fall into two broad categories. On one hand there are systems that use the memory for caching (databases, web or generic caching systems). On the other hand we have runtime systems in which garbage collectors use the memory for object allocation.

Among caching systems, DMBSes like MySQL or Oracle rely on data buffers for faster access to data stored on disk. Web caches like Squid use main memory caches for reducing request latencies and bandwidth usage. Memcached enables general purpose key-value caches for any type of applications. What all these systems have in common is that they use the main memory they have under control for decreasing latency. The supporting data structures and policies for caching systems are often quite simple: linear lists and hashmaps with some sort of LRU policy. Consequently, a memory reservation mechanism is straight forward to implement. The most complex representatives of this category are probably the database systems. As we have seen with MySQL/InnoDB, the cost of implementing an ALB module was modest. For all other caching systems, the approach will be similar.

As garbage collectors (GCs) of runtime systems come in very many flavours: copying, mark-sweep, generational, world-stop vs. incremental vs. concurrent, etc., the supporting data structures and policies are more complex than in the case of caching systems. Still, we have shown in this paper that implementing an ALB module for a given GC (i.e., PSGC) is possible with little coding overhead, as long as the semantics of the supporting data structures are understood.

Once a memory reservation mechanism is implemented for an application, it needs to be controlled. Either the application polls for changes in the reservation or it is externally notified. In either case, an ALB control thread must be added to the application.

6. Discussion

While ALB yields significant benefits, a valid concern is to what extent it generalizes to other databases or language runtime systems. A good way to frame this discussion is to distinguish between, on the one hand, the interface allowing runtime dynamic memory reallocation between database memory pools or JVM Heaps located in different virtual machines, and on the other hand, the particular implementation technique we have chosen.

In the first case, existing systems lack a specialized interface between the applications and the OS to coordinate memory use. The existing interfaces between OS and applications only expose basic memory operations, and missing is an API to allow an application to release explicitly identified memory pages, effectively giving the OS the information needed to make an informed choice of pages to reallocate from the database. Where the OS runs in a VM, this information could be propagated to the VMM improving its reallocation choice also. The interface needs to support the reverse direction also, allowing the application to signal its need for more memory when required.

We argue from our results that providing such an interface has tangible benefits in resource management, and also that the interface itself can be relatively application-agnostic – there is little in the interface we have designed that is specific either to MySQL/ InnoDB, OpenJDK, or to our ballooning implementation.

The second question is one of implementation, arising from the application's assumption that memory is dedicated to its use. In databases, a mechanism complementing the existing database memory manager(s) could use an interface such as ours to dynamically adjust the size of memory pools. In JVMs, such a mechanism could have a tighter integration with the garbage collector, up to the point where the notorious "java.lang.OutOfMemoryError" errors are handled by requesting memory from another VM rather than crashing.

We sidestep such a mechanism in order to minimize the changes needed to existing code bases, and instead use application-internal routines to "allocate" memory to release via the balloon. For us, this technique has worked remarkably well, and we feel ballooning as an implementation will generalize to many (though not all) buffer managers and language runtime systems. Where it does not work, an open question is whether an alternative implementation would be possible without significant redesign.

On the database side, an interesting question is what happens to query-plan optimization in the case of ALB. While the TPC-H workload we show in this paper did not exhibit changes in performance due a smaller buffer pool than that advertised to MySQL, there may naturally be query optimizers that could make better choices based on the current (actual) size of the buffer pool. Treating the buffer pool as a variable in the system rather than a constant might improve query optimization.

Even though the current speed of OpenJDK ALB operations is sufficient for many scenarios, we want to further investigate possible optimizations by extending the design of GCs so that ALB is a first class citizen. Determining the heap generation and space in which ALB memory should be ballooned-in or -out based on specific workloads is of particular interest.

Complementing the memory managers in more complex systems with a ballooning module can be challenging. While reserving certain memory in the application is possible, the semantics for the reservation are not clear. If databases or language runtime systems are made aware of the ballooning process, some of the possible side-effects of ALB could be avoided.

A requirement (as we point out Section 2.3) for ALB to be useful is the correlation of memory size and performance. We assume no specific memory-vs.-performance model in this work, since we agree that this is likely application-specific, and instead view this as a policy issue. Such models are an active research area: recent papers address precisely this problem in databases ([6, 20, 25]), assuming no changes to the database. Robust behavior of databases under load is an open problem both in research and in industry ([28]).

An implication of our work is that if server applications can be changed to both emphasize predictability and expose their cost functions (like the ideas presented in [7]), this will enable better correlation of resource requirements with performance, and further increase the effectiveness of ALB as a mechanism for reducing over-provisioning in virtualized environments. Future systems designed for virtualized environments and architected to operate a balloon can also introduce more flexibility on both sides of the application/OS boundary.

We decided not to discuss ballooning policy in the paper. The performance correlation mentioned above is only one reason why policy for ALB is a complex and open question. For instance, a database balloon driver may use different types of memory (buffer, working memory, result caches, etc.), each calling for different policies and with different implications for the application. There are also several policy types: arbitration of resources across applications and management of an application's internal demands both require mechanisms not yet available at the application level and which demand a richer interface between applications and the VM. ALB is, therefore, a first step in tackling this generic problem, but is also immediately applicable: static solutions like those proposed by [25] can also use ALB for dynamic reconfiguration.

7. Related Work

Effectively managing memory in VMMs is an important topic. In addition to ballooning, modern VMMs also employ other techniques.

Content-based memory de-duplication reduces memory consumption by copy-on-write sharing of identical pages across the VMs [8, 16, 29]. Orthogonal to ALB, deduplication can be used to further reduce memory pressure within a physical machine.

Heo et al. [9] use control theory to meet memory utilization targets when overbooking memory with ballooning in Xen-based consolidation environments. The approach requires measurable memory demands to drive the control loop. Autocontrol [21] also applies control theory to optimize resource assignment to VMs, focusing on CPU and disk bandwidth. ALB provides a mechanism which could extend Autocontrol's policy engine to RAM.

MEB [33] does dynamic memory balancing for VMs based on runtime statistics from the VM kernel in order to optimize performance of memory dependent applications. MEB uses OS-level ballooning in Xen to re-configure memory. Figure 1 shows that this is not applicable to applications which manage their own memory, though it might fare better with some systems like PostgreSQL which use the OS disk cache for this purpose.

Live VM migration can also be used to relieve memory pressure in an overcommitted physical machine [30]. Migration is complementary to our approach, but does not address, for example, the poor performance of a database provisioned with less memory than initially configured.

Alonso and Appel [1] observe that the traditional model of virtual memory working sets is not always efficient for garbage collected systems or for databases caches. They propose a "memory advice" server which can be used to adjust application working sets. ALB as a mechanism for dynamically adjusting the working sets of applications could be enhanced with such a monitoring and management service.

7.1 Language runtime performance in VMs

Khanna et al. [13] present a bin-packing approach for identifying the best way to consolidate services on hardware, based on the applications' requirements. CRAMM [32] enables garbage collected applications to predict appropriate heap sizes, making the case for dynamic reallocation of RAM. CRAMM would provide a policy framework to drive ALB's reallocation mechanism. Hertz et al. [10] also identify the need for dynamic reconfiguration of heap sizes for language runtimes (C# in their system), and provide a way of sharing memory among different runtimes, within the same OS, compared to ALB which achieves this among different applications in different VMs. Hines et al. [11] present policy framework (Ginkgo) which correlates application performance to its memory needs in order to satisfy capacity and performance constraints using runtime monitoring. They dynamically resize the JVM heap using a balloon process that allocates and deallocates memory through JNI.

Finally, a recent patent from VMware [15] describes ballooning in JVM by allocating Java objects. As discussed in Section 3.2, ALB takes the alternative approach to both Ginkgo and VMware by integrating ballooning with the garbage collector.

7.2 Databases on VMs

Most recent work on running databases in VMs has focused on configuring the VMs to optimize the behavior databases in multi-tenant scenarios. Soror et al. [25] tune database performance by controlling the underlying VM. They argue that if resources are distributed to databases regardless of actual load, performance will suffer relative to a load-aware allocation. In this approach, resource distribution is static but can nonetheless be improved with successive deployments. Their incremental load monitoring proposal can be combined with ALB to implement a truly dynamic tuning mechanism.

Ozmen et al. [20] optimize data layout on the storage layer by examining the load characteristics of each DBMS tenant, and is orthogonal to ALB. Their solution also works for non-VM DBMS.

Minhas et al. [17] show how to use the availability facilities of VMMs to provide fault tolerance in virtualized databases through state machine replication. This approach can be combined with ALB to allow a new replica of a database to receive more memory on start, improving the performance of the backup and avoiding further failures if the initial incident was due to lack of memory.

Curino et al. [6] looked at consolidation gains of DBs on VMMs and report up to 17:1 consolidation ratios for the analyzed workloads. They note that existing OS ballooning techniques for consolidating databases are useless as they are not integrated with the DB's buffer pools.

Microsoft's SQL Hyper-V/VM [22] permits dynamic memory configurations for SQL Server, but does not present any details on how this is achieved and how the database reacts to memory changes. In [18], SQL-VM is presented as a solution for resource (CPU, I/O and memory) sharing in multi-tenant database-as-a-service systems.

In contrast to white-papers and patents form the industry, we present a detailed technical description and reproducible performance analysis of ALB across several systems. Moreover, ALB is generic and can be extended to any applications managing their own memory (whether a buffer pool or runtime heap), whereas VMware focuses only on a JVM heap and HyperV only on the MS SQL database.

8. Conclusions

We demonstrate ALB as a mechanism to efficiently vary the memory available to two server applications: MySQL and OpenJDK. We have shown that resulting performance is no different to that of a statically configured application of the same size, with no need for restart with a new configuration. Ballooning creates memory pressure within the application itself forcing it to make more well-informed decisions on buffering I/O or garbage collection than either the OS or VMM can do on its behalf. In the case of database collocation, we showed that ALB can be used to react to changes in the workload such that system throughput does not degrade. This is achieved by on-the-fly buffer pool resizing rather then having an over-provisioned buffer pool.

ALB piggy-backs onto existing OS-level ballooning, creating an end-to-end solution for memory reallocation, coordinated across all three software layers: VMM, OS, and application. The changes required to implement ballooning are small and limited to the OS (and its balloon driver) and application itself.

We expect rigid server applications to evolve to support varying memory usage in the long term. However, ALB is a simple, efficient, readily implementable technique available now to vary memory usage in rigid applications on shared machines. It is an enabling mechanism for new avenues of research in dynamically provisioning memory in previously rigid applications.

References

[1] R. Alonso and A. W. Appel. An advisor for flexible working sets. In *Proc. of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 153–162, New York, NY, USA, 1990. ACM.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In *Proc. of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[4] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD Record*, 39(3):5–10, Feb. 2011.

[5] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *Proc. of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[6] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *Proc. of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 313–324, New York, NY, USA, 2011. ACM.

[7] J. Giceva, T. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. COD: Database/Operating System Co-Design. In *6th Biennial Conference on Innovative Data Systems Research*, CIDR '13. www.cidrdb.org, 2013.

[8] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Comm. of ACM*, 53(10):85–93, Oct. 2010.

[9] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In *Proc. of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, IM'09, pages 630–637, Piscataway, NJ, USA, 2009. IEEE Press.

[10] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, and J. E. Bard. Waste not, want not: resource-based garbage collection in a shared environment. In *Proc. of the international symposium on Memory management*, ISMM '11, pages 65–76, New York, NY, USA, 2011. ACM.

[11] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. Ryu, and M. Ben-Yehuda. Applications Know Best: Performance-Driven Memory Overcommit with Ginkgo. In *Proc. of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 130–137, Washington, DC, USA, 2011. IEEE Computer Society.

[12] InnoDB LRU Dump/Restore `http://www.percona.com/docs/wiki/percona-server:features:innodb_lru_dump_restore`, 29.11.11.

[13] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application Performance Management in Virtualized Server Environments. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, NOMS '06, pages 373 –381, Washington, DC, USA, April 2006. IEEE Computer Society.

[14] R. Love. *Linux Kernel Development, 3rd edition*. Addison-Wesley, 2010.

[15] R. Mcougall, W. Huang, and B. Corrie. Cooperative memory resource mamagent via application-level balloon. Patent Application US20110320682, 12 2011.

[16] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: enlightened page sharing. In *Proc. of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[17] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent High Availability for Database Systems. *PVLDB*, 4(11):738–748, Nov. 2011.

[18] V. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *6th Biennial Conference on Innovative Data Systems Research*, CIDR '13. www.cidrdb.org, 2013.

[19] Lawrence Berkeley National Lab: Nux toolkit. `http://acs.lbl.gov/software/nux/api/nux/xom/sandbox/XQueryBenchmark.html`, 5.01.12.

[20] O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-aware storage layout for database systems. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 939–950, New York, NY, USA, 2010. ACM.

[21] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proc. of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM.

[22] Running SQL Server with Hyper-V Dynamic Memory. `http://msdn.microsoft.com/en-us/library/hh372970.aspx`, 17.10.12.

[23] Saxon XSLT and XQuery Processor. `http://sourceforge.net/projects/saxon/`, 5.03.12.

[24] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: a benchmark for XML data management. In *Proc. of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 974–985. VLDB Endowment, 2002.

[25] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1):7:1–7:47, Feb. 2008.

[26] TPC-E. `http://www.tpc.org/tpce/`, 17.10.12.

[27] TPC-H. `http://www.tpc.org/tpch/`, 17.10.12.

[28] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2(1):706–717, Aug. 2009.

[29] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[30] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proc. of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.

[31] Xen Hypervisor 4.1. `http://xen.org/`, 29.11.11.

[32] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: virtual memory support for garbage-collected applications. In *Proc. of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 103–116, Berkeley, CA, USA, 2006. USENIX Association.

[33] W. Zhao, Z. Wang, and Y. Luo. Dynamic memory balancing for virtual machines. *SIGOPS Oper. Syst. Rev.*, 43(3):37–47, July 2009.