ABSTRACT

PHILAR, GAUTAM R. Improving Memory Reclamation Techniques in Hypervisors using Persistent Memory. (Under the direction of Dr. Vincent W. Freeh.)

Virtualization is a core technology that enables cloud computing. Hypervisors enable virtualization by providing an abstraction layer for physical compute, storage and network resources. In a virtual environment, physical memory of a system is also shared between the guest operating systems. Hypervisor allows users to power on virtual machines with a total configured memory that exceeds the memory available on the physical machine. This is called memory overcommitment. While this increases the consolidation ratio (VMs per machine) it can result in severe performance degradation during low memory conditions. Currently used memory reclamation techniques like Ballooning and Transparent page sharing are limited and slow. They tend to degrade performance of other VMs that are hosted on the same hypervisor.

Persistent Memories (PM) refer to a new class of hardware devices that promise to bridge the gap between low-latency volatile DRAM and high-latency persistent HDDs/SSDs. They are byte-addressable and can be directly connected to the memory bus of a processor. This enables the processor to access these devices through load/store operations just like physical memory. With their near-memory like latency characteristics, PM devices can be used effectively as an intermediate caching layer between fast main memory and slow hard disk drives.

In this dissertation, we propose the use of Persistent memory to alleviate performance issues in hypervisors during low memory conditions. The main contributions of this work are (a) Persistent memory simulator using RAM (b) KVM-extension that migrates pages to and from the PM Zone intelligently. To evaluate our proposal, we ran several micro and macro benchmarks and observed a performance improvement of 34% over Ballooning and 45% over Hypervisor based swapping to disk.
Improving Memory Reclamation Techniques in Hypervisors using Persistent Memory

by
Gautam R. Philar

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina
2015

APPROVED BY:

Dr. Edward Gehringer
Dr. Xipeng Shen

Dr. Vincent W. Freeh
Chair of Advisory Committee
DEDICATION

To my lovely wife Gabriela and my parents for their unwavering support and trust in me.
BIOGRAPHY

Gautam R. Philar is a systems software engineer with background and interests in Virtualization and Cloud computing. Before enrolling as a full-time graduate student at North Carolina State University, he worked in the Storage Area Network development team at NetApp in RTP. Prior to that, he completed a graduate program in Information Systems from Carnegie Mellon University. After graduation, he intends to pursue an opportunity with Avere Systems, a Pittsburgh based startup that is pioneering cloud bursting solutions.
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Vincent W. Freeh for his incisive guidance in the overall direction of this thesis. I would also like to thank Dr. Edward Gehringer and Dr. Xipeng Shen for taking time out of their busy schedules to accommodate me.
TABLE OF CONTENTS

LIST OF TABLES ................................................................. vii

LIST OF FIGURES ............................................................. viii

Chapter 1  Introduction ..................................................... 1

Chapter 2  Motivation, Hypothesis and Contribution .................... 5
  2.1  Motivation ............................................................... 5
  2.2  Hypothesis ............................................................. 8
  2.3  Contribution .......................................................... 9

Chapter 3  Qemu-KVM Architecture ......................................... 10
  3.1  Qemu Architecture .................................................... 12
  3.1.1  Threading model .................................................... 12
  3.1.2  libvirt layer ......................................................... 14
  3.1.3  virtio layer ........................................................ 14
  3.2  Kernel based virtual machine (KVM) ................................ 15
  3.2.1  Intel hardware assisted virtualization ......................... 15
  3.3  Interaction with underlying kernel .................................. 18

Chapter 4  Design and Implementation ..................................... 19
  4.1  Persistent Memory Simulator ......................................... 20
  4.1.1  Memory Zones ...................................................... 20
  4.1.2  Modified Memory map ............................................. 21
  4.1.3  Micro-benchmarks for PM simulator ........................... 22
  4.2  Qemu-Kvm extension .................................................. 24
  4.2.1  Page fault handling ................................................. 24
  4.2.2  Migrating pages from low priority queues ...................... 25
  4.2.3  Migrating entire low priority queues ........................... 26

Chapter 5  Evaluation and Results ......................................... 28
  5.1  Experimental Setup ................................................... 28
  5.2  Performance tests ..................................................... 29
  5.2.1  Baseline ballooning - S0 .......................................... 29
  5.2.2  Baseline hypervisor swap to disk - S1 ......................... 29
  5.2.3  Baseline hypervisor swap to PM - S2 ........................ 30
  5.2.4  Hypervisor swap of low priority pages to PM - S3 .......... 30
  5.2.5  Hypervisor swap of low priority queues to PM - S4 ....... 30
  5.2.6  Hypervisor swap of high priority queues to PM - S5 ...... 31
  5.3  Results and Explanation ............................................. 31
  5.3.1  Baseline Results ................................................... 31
  5.3.2  Hypervisor swap to PM Vs Disk .................................. 33
  5.3.3  Qos control ........................................................ 34
Chapter 6 Related Work ................................................................. 36
  6.1 PM aware filesystems ......................................................... 36
  6.2 PM programming ............................................................ 37

Chapter 7 Conclusion and Future Work ................................. 38

Bibliography ................................................................. 40
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Baseline system configuration</td>
<td>22</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>DRAM/PM Performance test configuration</td>
<td>22</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Performance comparison DRAM and PM</td>
<td>23</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Hardware Configuration</td>
<td>28</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Test scenarios</td>
<td>29</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Test scenarios Performance Results</td>
<td>33</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Kernbench Configuration</td>
<td>35</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>QoS performance comparisons</td>
<td>35</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Virtual memory mapping in a hypervisor</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Shadow Page table</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Hypervisor Memory Overcommitment</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Memory Ballooning</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Qemu processes</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Qemu Internals</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Intel VMX operations</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>VMM Handling</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>GVA to HPA conversion</td>
<td>18</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Modified Memory Map</td>
<td>21</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>DRAM/PM Baseline</td>
<td>23</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Low priority Qemu queues</td>
<td>27</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Ballooning and HV-swap baseline</td>
<td>31</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Comparing run-times for different scenarios</td>
<td>33</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Qos control</td>
<td>35</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>PMFS Vs Legacy FS</td>
<td>36</td>
</tr>
</tbody>
</table>
Server virtualization software enables several operating systems to run on a single physical server or host as self-contained virtual machines (VMs) isolated from each other yet sharing the host’s physical resources. This server virtualization software that runs directly on the physical hardware is called a Hypervisor or Virtual machine monitor (VMM).

The hypervisor intelligently manages physical resources like CPU, Memory, Network and Storage among guest operating systems. In this thesis, we are primarily interested in memory virtualization. It involves sharing system memory and dynamically allocating it to virtual machines. Virtual machine memory virtualization is similar to virtual memory support provided by modern operating systems. Processes see a contiguous address space that is not necessarily tied to the underlying physical memory of the system. The operating system maintains mappings of virtual page numbers to physical page frames in page tables. All modern CPUs include a memory management unit (MMU) and a translation lookaside buffer (TLB) to optimize virtual memory performance.
As shown in Figure 1.1[VMWare (2011)], to run multiple virtual machines on a single system, another level of memory virtualization is required. The MMU has to be virtualized in order to support multiple guest OSes. Each guest OS continues to control the mapping of guest virtual memory addresses to the guest physical memory addresses, but the guest OS cannot have direct access to the actual machine memory also called the host physical memory. The hypervisor is responsible for mapping guest physical memory to the actual machine memory in a structure called shadow page table. The hypervisor uses TLB hardware to accelerate guest virtual memory to host physical memory translation and avoid the tedious two level translation - guest virtual address → guest physical address → host physical address on every memory access. When the guest OS changes its virtual memory to physical memory mapping, the hypervisor updates the shadow page tables to enable a direct lookup.

The following terminology is used throughout this thesis with their relationships illustrated.
1. Guest virtual memory refers to the continuous virtual address space presented by the guest OS to applications running inside the virtual machine.

2. Guest physical memory refers to the memory visible to the guest OS running in the virtual machine.

3. Guest physical memory is backed by Host physical memory, which means the hypervisor provides a mapping from the guest to the host memory.

4. Guest level paging refers to memory transfer between the guest physical memory and the guest swap device and is driven by the guest OS. The memory transfer between host physical memory and the host swap device is referred to as Hypervisor swapping and is driven by the hypervisor.

As shown in Figure 1.2[VMWare (2011)], address translation between guest physical memory and host physical memory is maintained by the hypervisor using a per VM data structure called pmap. The hypervisor intercepts all virtual machine instructions that manipulate the hardware TLB contents or guest OS page tables. The actual TLB state is updated based on a separate shadow page table, which contains the guest virtual to host physical address mapping. The shadow page tables maintain consistency with the guest OS page tables and the pmap data structure. This reduces the two-level translation overhead because the hardware TLB will cache direct guest virtual address to host physical address translations read from the shadow page tables. The extra level of indirection of guest physical memory to host physical memory enables the hypervisor to easily remap a virtual machine's host physical memory to other devices in a manner that is completely transparent to the VM.

Hypervisors support overcommitment of physical resources thus enabling higher consolidation ratios. A hypervisor is said to be memory-overcommitted when the total configured guest memory size of all powered-on VMs exceeds the physical memory on the hypervisor. When a hypervisor is memory-overcommitted, it distributes physical memory fairly and efficiently among the powered-on VMs.
For instance in Figure 1.3[VMWare (2011)], hypervisor has only 4GB physical memory but is configured to support three virtual machines with 2GB guest physical memory each. Without memory overcommitment, only one VM can be run considering that each guest VM has some memory overhead. In order to effectively support memory overcommitment, the hypervisor must provide efficient host memory reclamation techniques. In this thesis, we propose the use of emerging Non-volatile persistent memory to alleviate some of the performance issues observed during memory reclamation in hypervisors.
2.1 Motivation

Memory overcommitment is an extremely important feature of a hypervisor. If it is not supported, the hypervisor needs to reserve enough host physical memory upfront to back all VMs’ guest physical memory. This reduces the consolidation ratio which is a measure of the number of VMs that can be configured on a physical server. A lower consolidation ratio means lower memory utilization. Hence, memory overcommitment and efficient reclamation techniques form an important feature set of a commercially viable hypervisor.

Currently, most hypervisors support the following memory reclamation techniques.

1. **Transparent Page Sharing**: When multiple VMs are running, some of them may have identical sets of memory content. For example, several virtual machines may be running the same guest operating system, have the same applications or contain the same user data. With TPS, hypervisor can reclaim the redundant copies and keep only one copy, which is shared by multiple VMs in the host physical memory. As a result, the total VM host memory consumption
is reduced and a higher degree of memory overcommitment is possible.

2. **Ballooning**: This is a technique where the hypervisor makes a guest operating system aware of the low memory status of the host. Due to the VM’s isolation, a guest OS is not aware that it is running inside a VM. It is certainly not aware of the states of other VMs running on the same host. When the amount of free host physical memory becomes low, none of the VMs will free guest physical memory because they cannot detect the host’s physical memory shortage. A balloon driver loaded into the guest OS polls the hypervisor to obtain a target balloon size. If the hypervisor needs to reclaim guest physical memory, it sets the proper target balloon size for the balloon driver, making it inflate by allocating guest physical pages within the virtual machine. In Figure 2.1[VMWare Academic Program(2013)], four guest physical

![Figure 2.1: Memory Ballooning](image)

...
them because no further memory accesses to those pages will be generated by the guest OS.

3. **Hypervisor swapping:** At VM startup, the hypervisor creates a separate swap file for each VM. If necessary, the hypervisor directly swaps out guest physical memory to the swap file to reclaim host memory. This technique is used as the last resort because it can severely degrade performance of all VMs hosted on a hypervisor.

All the techniques discussed up to this point suffer from the following limitations.

1. Limits on the reclaimed memory size.
2. Slowness in reclaiming memory.
3. Degraded performance impacting other VMs.
2.2 Hypothesis

We hypothesize that Hypervisor swapping used in conjunction with emerging Non-volatile persistent memory will perform better than existing memory reclamation techniques like TPS, Ballooning and hypervisor swapping to disk.
2.3 Contribution

1. RAM backed Persistent Memory Simulator.

2. KVM-extension that migrates pages between PM Zone and DRAM intelligently.

3. Performance tests that evaluate our proposals to improve memory reclamation in hypervisors.
Qemu stands for quick emulation. It is an open source machine emulator and virtualizer. Qemu has two operating modes:

1. Full system emulation: This mode provides full platform virtualization. In this mode, Qemu emulates processor and various peripheral devices. This is the mode used throughout this project.

2. User mode emulation: This mode provides application level virtualization using dynamic translation. In this mode, programs made for one type of CPU (e.g., ARM) can be run on a different machine (x86) but the operating systems should be the same. We do not use this mode in this project.

When operated in full system emulation mode (virtualizer), Qemu can achieve near native performance by executing guest operating system code directly on the host CPU using an in-kernel accelerator called Kernel based virtual machine (KVM). KVM executes some of the guest code natively, while continuing to emulate the rest of the machine. Additionally, Qemu uses a full software Memory Management Unit (MMU) to handle virtual to physical address translation at every memory access. A soft MMU is used to improve portability.

Qemu runs as a userspace process. An ISO or bootable image of the guest operating system is provided as an argument during launch. The host kernel schedules Qemu like a regular process. Mul-
multiple guests run alongside without knowledge of each other. Other user applications like browsers, email clients etc also compete for the same host resources as Qemu, although resource controls can be used to prioritize qemu processes. When a guest operating system shuts down, the Qemu process exits.

Figure 3.1[KVM Architecture Overview (2015)] shows 3 different guest OSes running in a Qemu environment.

Since Qemu system emulation provides a full virtual machine inside the Qemu userspace process, the details of processes running inside the guest operating systems are not visible to the host. So in Figure 3.1, details of processes running within the guest OSes are not available to the underlying Linux kernel (also referred to as the host). In a nutshell, Qemu allocates a slab of guest RAM at startup, the ability to execute code and emulated hardware devices, so that any operating system can run as a guest.
3.1 Qemu Architecture

3.1.1 Threading model

Qemu uses a hybrid model that combines event-driven programming with threads. Since an event loop alone is single threaded, it will not be able to utilize multiple cores. So it is used in conjunction with threading that provides dedicated threads to perform certain functions. During configuration, the number of virtual CPUs available to a guest OS can be specified. Each virtual CPU is emulated as a vCPU thread within the Qemu process. These vCPU threads execute simultaneously and have to be synchronized with a global mutex lock. A separate ioThread runs an event loop to process asynchronous events like network packets and disk I/O and dispatches it to worker threads.

Figure 3.2 [KVM Architecture Overview (2015)] shows the internal threading model of Qemu.

![Qemu Internals](image.png)

Figure 3.2: Qemu Internals
3.1.1.1 Event-driven Qemu core

Running a guest involves executing guest code, handling timers, processing I/O and responding to monitor commands. These are mostly asynchronous events that are handled by the Qemu core thread. The Qemu core thread runs an event loop that listens for these events and dispatches to event handlers. The event handlers can be dedicated worker threads that run independently on separate cores thus providing parallelism.

The Qemu core thread uses the `poll()` system call to track actionable events. During initialization, a set of file descriptors with bitmaps indicating events of interest is set. The `poll()` system call examines these file descriptors for specified events and returns when those events have occurred. File descriptors play an important role because files, sockets, pipes and various other resources are all file descriptors. Timers can also be tracked using `poll()`.

When a file descriptor becomes ready or a timer expires, the event loop invokes a callback that responds to the event. No blocking system calls or long-running computations are performed in the callback since this will impact the main event loop. If blocking system calls are required, they are dispatched as a work item to worker threads that execute them asynchronously.

3.1.1.2 Worker threads

Blocking system calls and long-running computations are not handled as callbacks. They are handled by dedicated worker threads that operate independent of core Qemu thread. The core Qemu thread places these type of requests on a queue. Worker threads take requests off the queue and execute them outside of core Qemu. They may perform blocking operations or long-running computations since they execute in their own threads and do not block core Qemu. Communication between worker threads and core Qemu happens through file descriptors. A file descriptor is added to the event loop. When the worker thread writes to the file descriptor, core Qemu thread detects it and invokes a callback that was registered earlier.

3.1.1.3 Guest code execution

Qemu leverages the KVM module to execute guest code natively. KVM takes advantage of hardware virtualization extensions present in modern Intel and AMD CPUs for safely executing guest code directly on the host CPU. vCPU threads start executing guest code and take away control from core Qemu thread. Any asynchronous event like READ or WRITE to emulated device registers causes a vCPU thread to suspend and control is returned to core Qemu. Signals are used sometimes to solve the problem of guest code hogging the CPU. A signal invokes a signal handler that can return...
control to core Qemu from a thread executing guest code. Core Qemu then runs its event loop and processes any pending events.

### 3.1.1.4 IOthreads

IOthreads are specialized threads that do not execute guest code or core Qemu event-loop. They are meant to wait for long-running work items that can be executed independently from core Qemu or guest code. These threads are meant to take advantage of multi-cores in an SMP environment.

### 3.1.2 libvirt layer

Libvirt is a framework that all commercial hypervisors implement so that any popular virtual management tool like virsh or virt-manager can be used to manage virtual machines deployed on those hypervisors. It provides major functionalities like VM Management, Remote machine support, Storage Management and Network interface management. KVM-Qemu uses the libvirt framework so that any admin using the above mentioned tools will easily be able to deploy VMs.

### 3.1.3 virtio layer

Just like the libvirt layer, the virtio layer provides a standard way to access devices found in virtual environments. Rather than having a per-OS or per-environment way of accessing physical devices in virtual environments, this layer provides an industry standard specification to adhere to.
3.2 Kernel based virtual machine (KVM)

KVM is a linux kernel module that allows a user space program like Qemu to utilize the hardware virtualization features of processors. In our test setup, KVM uses Intel's virtualization technology (VT-x). While Qemu acts as the VMM, the kvm module needs to be loaded in-order to leverage hardware enabled acceleration. Qemu communicates with kvm as follows:

```c
open("/dev/kvm")
ioctl(KERN_CREATE_VM)
ioctl(KERN_CREATE_VCPU)
for (;;) {
    ioctl(KERN_RUN)
    switch (exit_reason) {
        case KVM_EXIT_IO:
        case KVM_EXIT_HLT:
    }
}
```

As stated earlier, KVM is an interface to the actual hardware support. In the next section, we explore Intel's VT-x features.

3.2.1 Intel hardware assisted virtualization

Soft-MMU provides portability but has performance problems. Maintaining consistency between guest page tables and host page tables through software on every memory access is a very CPU intensive operation. Any memory access in a guest OS that changes its page table entries will need to be reflected in the shadow page tables maintained by the host kernel. This consistency is required in order to maintain a coherent view of the hardware TLB that is utilized by the processor for fast virtual address to physical address translations.

Processor support for virtualization is provided by a form of processor operation called VMX operation. There are two types of VMX operations:

1. VMX root operation: A VMM will run in VMX root operation.
2. VMX non-root operation: A guest OS will run in VMX non-root operation. Processor behavior is restricted and modified to facilitate virtualization. Certain instructions and events cause VM exits to the VMM, providing the VMM with the ability to retain control of processor resources.
Transitions between these two operations are called VMX transitions. Transitions into VMX non-root operation are called VM entries and transitions from VMX non-root to VMX root are called VM exits.

1. VMM discovers support for VMX operations during bootup and enters VMX root operation by executing the VMXON instruction.

2. VMM then launches guest VMs using VMLAUNCH and VMRESUME instructions which causes the VM entry transition. At this point guest OS starts executing natively on the processor.

3. A VM exit transition transfers control back to the VMM. The VMM can take appropriate action to the cause of the VM exit and then return to the VM using a VM entry.

4. VMM leaves VMX operation by executing the VMXOFF instruction.
VMM configures a data structure called virtual-machine control structure (VMCS) for each logical processor (vCPU) per VM that it supports. This data structure is used to track VMX transitions. At any given time, at most one of the active VMCSs is the current VMCS. The VMCS data structure is used to maintain context data like guest-state and host-state which is used to load the processor state during VMX transitions.

VMX operations enable guest OSes to run directly on the hardware but that may still create excessive VMX transitions that impact performance. A way to reduce these transitions especially during paging is through Extended page tables (EPT). Using EPT allows guest OS to maintain control over legacy guest page tables and while VMM controls the EPT. The processor uses both these tables almost concurrently as follows:

Guest VA → Guest PA from Guest PT + EPT Base register → Host PA.
3.3 Interaction with underlying kernel

KVM leverages the standard Linux operating system as a base. It runs as a kernel extension, so it can easily make use of the Linux operating system's advanced support for x86 hardware. We have used this to our advantage by tweaking the memory manager in the underlying Linux kernel.
The central idea of this thesis is to demonstrate the use emerging byte-addressable persistent memory (PM) to address performance issues encountered during memory reclamation in virtualized environments. In chapters 1 and 2 we discussed some of the existing memory reclamation techniques and their major drawbacks. In this chapter, we propose a solution that leverages persistent memory (PM) to alleviate performance bottlenecks during low memory conditions in hypervisors.

New device technologies like phase-change-memory (PCM), spin-transfer-torque RAM and memristors provide persistent storage at near DRAM latencies. These technologies are collectively termed as persistent memory (PM). Data can be accessed directly through load/store instructions rather than through block I/O requests. In this research, we simulate RAM-backed PM and swap out pages to it *intelligently* during low memory condition. For prototyping, we have modified Linux kernel 3.7.1 to support our functionalities.
4.1 Persistent Memory Simulator

In designing a PM simulator, we had two design options.

1. Build the simulator as a loadable kernel module that created a RAM-backed /dev/pcm0 when it was loaded. This involved using IOCTLS in Qemu to initialize and communicate with it.

2. Build the simulator as a KVM-extension. This meant no changes to Qemu code and limited changes to the memory management code in the base kernel.

We chose the second option since it provides a clean way to isolate and easily integrate our simulator code with the proposed memory reclamation algorithm.

4.1.1 Memory Zones

The BIOS conveys the system memory map to the linux kernel through an E820 table. Each physical address range is associated with an address type that indicates how the physical address range is supposed to be used by the OS. We added a new address range type called AddressRangePM. This address range will be used to create a ZONE_PM as described in the next section. This modification enables the OS to distinguish between system RAM and part of RAM that is used to simulate persistent memory.

In a Non Uniform Memory Access (NUMA) architecture, memory is arranged into banks that incur a different cost to access depending on their distance from the processor. Each bank is referred to as a node and is tracked by struct pg_data_t in the linux kernel code. In our case, the test setup is Symmetric Multi-processor (SMP) environment. Each node is divided into a number of blocks called zones, which represent ranges within memory. With x86 architecture, the following zones are available:

1. ZONE_DMA - First 16MiB of memory
2. ZONE_NORMAL - 16MiB to 896MiB (Kernel memory)
3. ZONE_HIGHMEM - 896MiB to end (User memory)

Each zone is represented by struct zone_t. The system's memory is divided into fixed-size chunks called page frames that are 4096 bytes in size. Each physical page frame is represented by struct page and stored in global array called mem_map at the beginning on ZONE_NORMAL i.e start of kernel memory. To simulate PM, we introduce a new zone called ZONE_PM by splitting ZONE_HIGHMEM.
We introduce two global variables that are initialized along with the boot memory allocator. `pmstart_pfn` and `pmend_pfn` are initialized to create a new zone within ZONE_HIGHMEM. The entire address range `AddressRangePM` is in ZONE_PM.

### 4.1.2 Modified Memory map

Our test setup has 8GB physical memory. After creating ZONE_PM, the final memory map looks as follows:

![Modified Memory Map](image)

**Figure 4.1:** Modified Memory Map

1. ZONE_DMA - First 16MiB of memory
2. ZONE_NORMAL - 16MiB to 896MiB (Kernel memory)
3. ZONE_HIGHMEM - 896MiB to 5956MiB (User memory)
4. ZONE_PM - 5956MiB to end (Persistent memory)
4.1. PERSISTENT MEMORY SIMULATOR

Table 4.1: Baseline system configuration

<table>
<thead>
<tr>
<th>System</th>
<th>Quad core, 3.00 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM/PM</td>
<td>8GiB</td>
</tr>
<tr>
<td>Base OS type</td>
<td>64-bit Linux</td>
</tr>
<tr>
<td>Guest OS type</td>
<td>64-bit Ubuntu 14.04 LTS, 2 vCPUs</td>
</tr>
<tr>
<td>Guest DRAM/PM</td>
<td>4 GiB</td>
</tr>
<tr>
<td>READ Latency</td>
<td>200ns</td>
</tr>
<tr>
<td>WRITE Latency</td>
<td>250ns</td>
</tr>
</tbody>
</table>

Table 4.2: DRAM/PM Performance test configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Workload</th>
<th>Footprint</th>
<th>Baseline VM</th>
<th>Guest OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS</td>
<td>Bubble sort</td>
<td>1024 MB</td>
<td>4096 MB</td>
<td>64-bit Ubuntu</td>
</tr>
<tr>
<td>QS</td>
<td>Quick sort</td>
<td>1024 MB</td>
<td>4096 MB</td>
<td>64-bit Ubuntu</td>
</tr>
</tbody>
</table>

In our current setup, Page frames from 2096640 to 2097152 are used to simulate PM that is 2 GB in size.

4.1.3 Micro-benchmarks for PM simulator

For micro-benchmark performance testing, we compare performance of two popular sorting algorithms with large datasets in DRAM and simulated Persistent Memory. For PM tests, we simulate asymmetric READ and WRITE latencies by adding an artificial delay of 200ns for READs and 250ns for WRITEs. Our baseline system configuration is as shown in Table 4.1

4.1.3.1 Experimental setup

In our experiments, we compare performance of modified in-memory Quicksort and Binary sort, running in virtualized environment with the data being in DRAM and simulated Persistent Memory. Each workload is executed with a 1GiB data set that is initially read into memory.

Table 4.2 shows our test configuration. The data is initially read from HDD into DRAM and simulated PM. Run-times are measured starting from after the data has been read into memory. All subsequent READs and WRITEs are to the in-memory data structure used for sorting. All access to disk is ignored. qsort and bsort are executables running in the guest VM which is configured with enough memory. We try to limit the maximum memory footprint of both applications to 1024MB. The purpose of this benchmark is to establish the basic difference in performance between DRAM and persistent memory which is supposed to have DRAM-like latencies.
4.1.3.2 Results

In-memory binary sort and quick sort can be considered to be CPU-intensive as well as memory-intensive. Initially building up the data set in memory ensures that all further activity is limited to memory and there is no I/O activity. Our goal is to prevent any unwanted disk-latencies to creep into our analysis. After the initial burst of paging activity that is common to both scenarios, all further processing is limited to the processors and memory. Persistent memory will be attached to the memory bus and can be accessed by the processor just like a DIMM module. So simulating PM with RAM-backed pages is an ideal substitute.

The near-DRAM latencies provided by PM make it ideal for use as an alternative tier in the current memory hierarchy. We believe that a PM layer can be used as a cache between DRAM and secondary memory systems like SSD/HDDs to reduce latency incurred during paging. Pages are migrated between the PM layer and DRAM depending on the memory pressure.
4.2 Qemu-Kvm extension

In this section, we discuss the design and implementation details of a KVM-extension that is used to swap pages to and from the ZONE_PM. When there is memory pressure, pages are swapped into the ZONE_PM. Under normal load, pages backed by DRAM are used.

4.2.1 Page fault handling

Qemu enables the guest OS to execute natively on the CPU using KVM and VMX extensions. When Qemu is initially launched with a guest OS, it calls `malloc()` and allocates memory for the VM. But `malloc()` does a lazy allocation. It does not reserve physical backing store for the `malloc'd` memory. So as the guest OS begins executing, it causes a page fault which causes a VM exit operation back to the VMM. Qemu leverages the underlying kernel's page fault handler to allocate a physical page and update the page tables. The code flow is as follows:

```c
do_page_fault();
handle_mm_fault();
pmd_alloc();
ppte_alloc();
handle_ppte_fault();
do_no_page();
alloc_page() {
    struct page_t page;
    page = page_allocator();
    if (Qemu_low_pri_process) {
        enqueue(low_pri_queue, page);
    }
}
```

Initialize a global queue to track the physical frames being allocated to low priority Qemu processes. These frames will not be added to LRU lists. When a page is not present in the EPT, there is a page fault and the page fault handler is invoked. As part of allocating a new page to this process, insert this page in the global queue if this is a low pri Qemu process. Since at boot time, we have created a new ZONE_PM, page allocation will not happen from this region.
4.2.2 Migrating pages from low priority queues

Memory pressure builds up as processes use up available free page frames. This section focusses on a new way to select old pages that can be freed and invalidated for new uses before physical memory is completely exhausted. Currently, page replacement policy does not distinguish between processes. Any pages belonging to any process can be reclaimed. We modify this by introducing a low priority queue that tracks pages allocated to low priority Qemu processes. LRU-like algorithm is then run on this queue to reclaim pages.

During system startup, a kernel thread called \texttt{kswapd} is started from \texttt{kswapd_init()}, which continuously executes the function \texttt{kswapd()}. We modify this function as follows:

```c
static int kswapd(void p) {
    if (!zone->balanced) {
        move_inactive_pages_to_pm(&zone);
    }
}

void move_inactive_pages_to_pm(struct zone_t zone) {
    struct page_t page;
    spin_lock_irq(zone->lpr_i_lock);
    while (size(low_pri_queue) > 0 && pages_freed) {
        page = dequeue(low_pri_queue);
        if (page->is_locked() && page->PG_launder) {
            wait_on_page(page->page_id);
            continue;
        }
        if (page->PG_dirty) {
            page->PG_launder = 1;
            page_PG_dirty = 0;
            writepage(page->page_id);
        }
        copy_to_pm(page);
        mmu_notifier_modify_pte();
        spin_lock_irq(zone->lru_lock);
        refill_inactive_zone(page);
        spin_unlock_irq(zone->lru_lock);
        pages_freed--;
    }
}
```

25
4.2. QEMU-KVM EXTENSION

CHAPTER 4. DESIGN AND IMPLEMENTATION

} 
spin_unlock_irq(zone->lpri_lock);
}

\textbf{kswapd} is woken by the physical page allocator when \textit{pages_low} number of free pages in a zone is reached. It frees up a certain number of pages and reschedules itself until it is woken up again. This process continuous until the number of free pages reaches \textit{pages_high} for that zone. Under extreme memory pressure, processes will do the work of \textit{kswapd} synchronously. \textit{kswapd} traverses through the \textit{low_pri_queue} global structure and performs the following actions:

1. If the page is locked and PG_launder bit is set, it means this page is currently locked by the process for I/O, so it is skipped. It is put on the \textit{wait_queue} of this zone and will be moved to \textit{ZONE_PM} in the next iteration.

2. If the page is dirty and the PG_dirty bit is set, the contents are immediately flushed to the backing store by calling writepage().

3. Finally, if it is not a dirty page or has no process referencing it, the page is migrated to \textit{ZONE_PM}. It is then moved to the \textit{Inactive LRU list}, for \textit{kswapd} to find it there and move it to the free list. The corresponding pte is updated to reflect the new mapping.

4.2.3 Migrating entire low priority queues

Instead of migrating a set number of pages in each iteration, we migrate all pages of low priority Qemu processes to \textit{ZONE_PM} and release those pages in DRAM for consumption by higher priority Qemu processes. In order to do this, we maintain a list of \textit{low_pri_queues} for all the low priority Qemu processes in the system.

Everytime \textit{kswapd} runs, it traverses through this list migrating all the pages belonging to a particular low priority process to \textit{ZONE_PM}. This proceeds until the list is empty which indicates that all low priority processes have been migrated. Consider Figure 4.3, there are four low priority Qemu processes running alongside higher priority Qemu processes. The first time \textit{kswapd} runs in a low memory situation, it moves pages 232 and 123 to \textit{ZONE_PM}. In the second run, it moves pages 98, 34 and 12, thus freeing these frames for use by higher priority Qemu processes. The difference between this and the previous approach is that here we are moving all the frames being used by a particular process during a single run instead of the piece-meal approach in the previous section.

As the system transitions to normal memory conditions, pages are swapped back from \textit{ZONE_PM} to DRAM. \textit{kswapd} checks if the number of free pages equals the \textit{high_pages} count for the zone. If
so, it migrates pages back from ZONE_PM to DRAM and updates the page tables.
CHAPTER

5

EVALUATION AND RESULTS

5.1 Experimental Setup

We integrate the PM simulator with our KVM extension and test it on Linux base kernel 3.7.1. The hardware configuration is shown in Table 5.1:

We configure guest operating systems using KVM control groups (cgroups) in order to limit physical resources allocated to a particular guest OS and maintain a priority order among the guest OSes. Using cgroups provides us an ability to space reserve physical memory upfront in some of

<table>
<thead>
<tr>
<th>Table 5.1: Hardware Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System</strong></td>
</tr>
<tr>
<td>DRAM</td>
</tr>
<tr>
<td>PM</td>
</tr>
<tr>
<td>Base OS type</td>
</tr>
<tr>
<td>Guest OS type</td>
</tr>
<tr>
<td>Qemu-KVM version</td>
</tr>
<tr>
<td>READ Latency</td>
</tr>
<tr>
<td>WRITE Latency</td>
</tr>
</tbody>
</table>
Table 5.2: Test scenarios

<table>
<thead>
<tr>
<th>Config Name</th>
<th>Test details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Baseline ballooning</td>
</tr>
<tr>
<td>S1</td>
<td>Baseline Hypervisor swap to disk</td>
</tr>
<tr>
<td>S2</td>
<td>Baseline Hypervisor swap to PM</td>
</tr>
<tr>
<td>S3</td>
<td>Hypervisor swap of low priority pages to PM</td>
</tr>
<tr>
<td>S4</td>
<td>Hypervisor swap of low priority queues to PM</td>
</tr>
<tr>
<td>S5</td>
<td>Hypervisor swap of high priority queues to PM</td>
</tr>
</tbody>
</table>

The purpose of this configuration is to establish a baseline with a currently used memory reclamation technique called ballooning. As explained in a previous section, ballooning is a popular memory reclamation technique used by Qemu-KVM, VmWare-ESX and Xen. We configure both static and dynamic memory configurations and enable the balloon driver in every guest operating system. We measure the runtimes for LKC and Pbzip2 and use the average values in our analysis.

5.2.2 Baseline hypervisor swap to disk - S1

The purpose of this configuration is to establish a baseline with a currently used memory reclamation technique called hypervisor swapping. As explained in a previous section, HV-swapping is a last-
resort memory reclamation technique used by Qemu-KVM, VmWare-ESX and Xen. We configure both static and dynamic memory configurations and disable the balloon driver in every guest operating system. During low memory conditions, the hypervisor resorts to uncooperative swapping of pages to HDD. We measure the runtimes for LKC and Pbzip2 and use the average values in our analysis.

5.2.3 Baseline hypervisor swap to PM - S2

The purpose of this configuration is to establish a baseline with a currently used memory reclamation technique called hypervisor swapping. As explained in a previous section, HV-swapping is a last-resort memory reclamation technique used by Qemu-KVM, VmWare-ESX and Xen. We configure both static and dynamic memory configurations and disable the balloon driver in every guest operating system. During low memory conditions, the hypervisor resorts to uncooperative swapping of pages to Persistent-Memory(PM) Zone. We measure the runtimes for LKC and Pbzip2 and use the average values in our analysis.

5.2.4 Hypervisor swap of low priority pages to PM - S3

The purpose of this configuration is to test Hypervisor swapping of low priority pages to ZONE_PM. As explained in a previous section, HV-swapping is a last-resort memory reclamation technique used by Qemu-KVM, VmWare-ESX and Xen. We configure both static and dynamic memory configurations and disable the balloon driver in every guest operating system. We use croups to configure priorities for each guest OS. Guest OSes are configured as either High or low priority. During low memory conditions, the hypervisor initially swaps pages belonging to low priority guest OSes to PM. We measure the runtimes for LKC and Pbzip2 in High priority guest OSes and use the average values in our analysis.

5.2.5 Hypervisor swap of low priority queues to PM - S4

The purpose of this configuration is to test Hypervisor swapping of low priority queues to ZONE_PM. We configure both static and dynamic memory configurations and disable the balloon driver in every guest operating system. We use croups to configure priorities for each guest OS. Guest OSes are configured as either High or low priority. During low memory conditions, the hypervisor initially swaps entire low priority queues belonging to low priority guest OSes to PM. We measure the runtimes for LKC and Pbzip2 in High priority guest OSes and use the average values in our analysis.
5.2.6 Hypervisor swap of high priority queues to PM - S5

The purpose of this configuration is to test Hypervisor swapping of high priority queues to ZONE_PM. We configure both static and dynamic memory configurations and disable the balloon driver in every guest operating system. We use *cgroups* to configure priorities for each guest OS. Guest OSes are configured as either *High* or *low* priority. During low memory conditions, the hypervisor initially swaps entire high priority queues belonging to high priority guest OSes to PM. We measure the runtimes for LKC and Pbzip2 in High priority guest OSes and use the average values in our analysis.

5.3 Results and Explanation

In the first part of our analysis, we establish a baseline to get a sense of current reclamation techniques. We then run the same tests with our KVM-extension and compare run-times to the baseline.

5.3.1 Baseline Results

To establish a baseline we ran static and dynamic tests and measured the run-times for LKC and Bzip2. In the static environment, we configured 2 cgroups with 4Gb memory capacity and 1 Active Linux guest operating system. For dynamic environment, we configured 2 cgroups with 4Gb each and 3 Active Linux guests. LKC and Bzip2 were executed in the active guest OSes. The first set of tests were conducted with Ballooning enabled on all guest OSes. The second set of tests were conducted with ballooning disabled, so that hypervisor swapping to disk would be used. From the results, it is apparent that Ballooning performs better than hypervisor swapping to disk. This is reasonable because the hypervisor resorts to un-cooperative swapping of guest OS pages. Since the hypervisor does not have any means to peek into the guest page tables, it runs an LRU algorithm on the host.

![Figure 5.1: Ballooning and HV-swap baseline](image)
page tables to find the next best candidate to swap out to disk. This can result in unnecessary swapping which results in degraded performance.
Table 5.3: Test scenarios Performance Results

<table>
<thead>
<tr>
<th>Config Name</th>
<th>Linux Kernel Compile</th>
<th>Pbzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline ballooning(S0)</td>
<td>72m 53.83s</td>
<td>15m 46.8s</td>
</tr>
<tr>
<td>Baseline Hypervisor swap to disk(S1)</td>
<td>100m 56.22s</td>
<td>22m 1s</td>
</tr>
<tr>
<td>Baseline Hypervisor swap to PM(S2)</td>
<td>63m 41.09s</td>
<td>12m 23s</td>
</tr>
<tr>
<td>Hypervisor swap of low priority pages to PM(S3)</td>
<td>52m 48s</td>
<td>11m 18s</td>
</tr>
<tr>
<td>Hypervisor swap of low priority queues to PM(S4)</td>
<td>48m 11.4s</td>
<td>9m 41s</td>
</tr>
<tr>
<td>Hypervisor swap of high priority queues to PM(S5)</td>
<td>55m 36s</td>
<td>12m 42s</td>
</tr>
</tbody>
</table>

5.3.2 Hypervisor swap to PM Vs Disk

In this set of tests, VMs were prioritized into High priority and Low priority using the cgroups feature of Qemu-KVM. Four high priority VMs were configured alongside four low priority VMs.

In static setup, one active Linux guest is configured to run while space reserving the other guest OSes using cgroups. This forces the guest OS to encounter low memory condition from the outset. In dynamic setup, multiple guest OSes run simultaneously and can encounter low memory condition at different points in time. Dynamic setup is akin to real-world mode where multiple applications running in different guest OSes can impact each other's performance if low-memory condition is encountered on the hypervisor. Each test was executed 5 times and average was used in the analysis.

Figure 5.2: Comparing run-times for different scenarios
5.3. RESULTS AND EXPLANATION

5.3.2.1 Linux kernel compile

LKC compiles and builds a linux kernel from source by reading the entire source directory into memory. Analysing the run-times, we see that Hypervisor swapping to disk has the worst performance. This is reasonable because un-cooperative paging at the hypervisor layer impacts guest VM performance. When swapping to ZONE_PM is enabled, we notice a performance improvement of 34% over Ballooning and close to 45% over Hypervisor based swapping to disk. The best improvement is seen with hypervisor swapping of low priority queues to ZONE_PM. By swapping out low priority process’ pages, we enable high priority processes to run faster as is confirmed by their lower run-times.

5.3.2.2 Pbzip2

Pbzip2 is a parallel implementation of bzip2. It performs compression on an entire Linux source tree using multiple threads. The execution times for both static and dynamic environments are as shown in the figures. They follow a similar pattern to LKC. Hypervisor swapping of low priority queues to ZONE_PM performs better than all other configurations with performance gains of 38% over ballooning and 56% over Hypervisor swapping to disk.

5.3.3 Qos control

Directly attached Persistent-memory devices can be used to provide quality of service guarantees to high priority guest operating systems. Providing Qos guarantees during memory reclamation is a way to ensure predictable performance for workloads that are deemed important. Currently, using cgroups, admins can prioritize a set of tasks i.e VMs running on a hypervisor. However, as the system reaches low memory condition, the performance of all the VMs becomes unpredictable. Even the high priority VMs are impacted by intensive memory reclamation techniques running in the background. Using PM that can be dynamically sized during such a situation provides us with an option to guarantee a certain level of quality.

For these tests, we use kernbench. Kernbench runs the kernel compile in a number of different ways. It cleans and primes a kernel tree with a make defconfig. Then it reads all the kernel source to cache it in RAM. Then it performs a number of different kernel compiles by varying the loads across the cores. This benchmark can be both CPU and memory intensive. The S4 configuration is setup for this test. Ballooning and HV-swap to disk are disabled. The hypervisor swaps low priority queues to ZONE_PM and we measure run-times for the following 4 types of workloads. Each test was executed 5 times and the average run-times of High Priority processes were collected.
Table 5.4: Kernbench Configuration

<table>
<thead>
<tr>
<th>Config Name</th>
<th>Test setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>2 High Priority, 4 Low priority VMs</td>
</tr>
<tr>
<td>C2</td>
<td>3 High Priority, 3 Low priority VMs</td>
</tr>
<tr>
<td>C3</td>
<td>4 High Priority, 2 Low priority VMs</td>
</tr>
<tr>
<td>C4</td>
<td>5 High Priority, 1 Low priority VMs</td>
</tr>
</tbody>
</table>

Table 5.5: QoS performance comparisons

<table>
<thead>
<tr>
<th>Tests</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-times in secs with PM</td>
<td>1084.2</td>
<td>1120.6</td>
<td>1316.5</td>
<td>1628.4</td>
</tr>
<tr>
<td>Run-times in secs with Balloon driver</td>
<td>1412.6</td>
<td>1562.6</td>
<td>1633.8</td>
<td>1714.8</td>
</tr>
<tr>
<td>Run-times in secs with HDD</td>
<td>1620.3</td>
<td>1739.6</td>
<td>1862.6</td>
<td>1914.3</td>
</tr>
</tbody>
</table>

Using the KVM-extension which swaps out low priority queues to ZONE_PM, we notice predictable performance with all configurations. With Balloon driver and swapping to HDD, as the number of high priority VMs increases, there can be more contention for memory. Since the number of physical pages allocated to low priority VMs that are swappable is less than pages allocated to high priority VMs, with low memory situations we observe significant performance degradation. On the other hand, with PM, even when there is memory contention, performance degrades slightly but with predictability.

Figure 5.3: Qos control
6.1 PM aware filesystems

Figure 6.1: PMFS Vs Legacy FS
This area of research explores the impact of load/store accessible Persistent Memory on system software design. It proposes a new filesystem design that exploits the PM’s byte-addressability by avoiding overheads of block drivers by enabling direct PM access by applications. SSDs are low latency block devices that are accessed through block drivers. Since they do not have a byte-addressable interface, this is probably the best way to access these types of devices. But in case of PM, block drivers pose an additional layer of abstraction that is an overhead. PM aware filesystem avoids these block layers and enable direct PM access by applications with memory mapped I/O. By mapping PM pages directly into an application's address space, the overhead of copying into kernel memory and then to the process's address space is avoided.

### 6.2 PM programming

Legacy applications are used to depending on filesystems to provide consistency in the event of any kind of failure. But in the case of a PM-aware filesystem, applications flush data directly to PM and will need to have access to libraries that effectively provides open/close/read and write semantics that filesystems provide. So these libraries provide a way to expose PM devices to applications circumventing the existing path through block device interfaces. This happens in conjunction with a PM aware filesystem. The application memory-maps PM pages through a PM-aware filesystem and directly does loads/stores to the PM region. But the application developer will then have to directly manage how data structures are laid out on the PM device and will also have to do failure handling. That's where these PM libraries provide an interface to handle memory allocation or transactional updates.
Hypervisor memory reclamation is an overlooked area in virtualization. Ballooning, Transparent Page Sharing and Hypervisor based swapping to disk are limited and slow. As consolidation ratio increases in data centers, hypervisors need to have robust memory reclamation techniques that perform well and provide quality of service. Newly introduced Persistent Memory devices provide the ideal caching layer to accomplish this goal. They provide persistence at memory-like latencies. As these new types of persistent memory devices become more mainstream, it becomes necessary to think about their effects on existing system software. Completely overhauling legacy software stack by introducing new PM aware filesystem or enabling applications to access PM devices directly may seem like feasible ideas in the long run. But in the near future, the best way to leverage these new devices is in the form of a hierarchical caching layer positioned in between main memory and the hard disk drive. To exploit this new layer, we propose some tweaks to the memory management in the form of a KVM-extension. We observe noticeable performance gains of 34% over Ballooning and 45% over hypervisor based swapping. Our proposal also provides a way to ensure quality of service to other higher priority VMs in case of memory pressure.

A future area of research will be to explore if this persistent memory area can be leveraged for more than just swapping pages when there is memory pressure. Dynamically resizing the PM zone
provides advantages to other memory intensive subsystems in the kernel. The filesystem meta-data can be stored in the PM zone and this region can be expanded and contracted by migrating pages to and from the disk. Another useful application can be as a hierarchical page cache. Currently pages in the page cache or buffer cache do not have priorities. Based on existing page replacement techniques, any page can be swapped out. By using a multi-tiered page cache pages can be migrated between tiers intelligently.
BIBLIOGRAPHY


