Karlsruher Institut für Technologie

Towards Power-Aware Memory for Virtual Machines

Diplomarbeit
von

Max Laier

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweigutachter:	Prof. Dr. Hartmut Prautzsch
Betreuender Mitarbeiter:	Dipl.–Inform. Jan Stöß

Bearbeitungszeit: 1. Mai 2009– 30. November 2009

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 30. November 2009

Max Laier

Abstract

The current trend towards hardware consolidation, using full machine virtualization techniques, demands more and more powerful host systems with huge amounts of main memory. To operate such systems economically, power management for the memory system is becoming an important issue. Previous work has only investigated power management for traditional operating systems.

This thesis, for the first time, analyzes the specific challenges and opportunities of the virtualization environment to provide memory power management. We adapt previously described techniques for memory power management in traditional operating systems to the hypervisor in a virtualization environment. In addition, we describe a number of additional techniques—specific to our environment—that help to save additional energy. We show that a set of relatively minor changes to the memory management can easily halve the static energy consumption in the memory system.

Our work is based on pure virtualization with no changes to the guest operating systems. Thus, a hardware consolidation setup can collect the benefit of our memory power management for legacy guest systems. For our evaluation, we use the latest, hardware-assisted, virtualization techniques, which provide almost raw hardware performance, and demonstrate that our changes do not impact on the performance of the virtualization.

Acknowledgments

I would like to thank the System Architecture Group at the University of Karlsruhe for making this thesis possible. In particular, I thank Prof. Dr. Frank Bellosa and Jan Stöß. I am also thankful to James McCuller for his help on the hardware setup. In addition, I am thankful to the support technicians at MSI who provided us with a number of custom BIOS builds in order to try to switch off the memory interleaving—eventhough these efforts never paid off.

I thank my family for the moral support, especially during the last few weeks, and my brother—Moritz—and my uncle—Michael—for their review of this thesis.

Contents

Chapter 1

Introduction

In recent years there has been an increasing trend towards virtualization. Virtualization provides an easy way to fully use more powerful hardware. It allows the operator to run several tasks, previously running on many small machines, on a single, more powerful machine. All major chip manufacturer provide specially equipped processors to support virtualization. As more and more virtual systems can be run on a single hardware system, these systems must be equipped with more and more main memory. Previous work has found that the main memory can easily account for up to 70% of the total energy consumption of the system. In this situation, power-management is becoming increasingly important. Power-management does not only reduce the energy bill for running the hardware itself, but—by decreasing the heat dissipation of the components—it also reduces the required cooling and increases lifetime and reliability.

This thesis will investigate memory power-management with the specific challenges of virtualization. To our knowledge, no previous work exists that is concerned with power-management of the memory system in the context of virtualization. Previous work does exists that proposes memory power-management for traditional operating systems. In this thesis we adapt the findings therein to our environment of a hypervisor running several virtual machines. The goal is to save energy in the memory system. This thesis focuses on the static energy consumption. The static energy is that required to keep the memory modules operational—in contrast to the dynamic energy that is required to read from and write to the modules. The static energy can be reduced by switching memory devices into a lower power state while they are not used by the currently running virtual machine. We will outline a number of changes to the memory allocation system that allow us to maximize the number of unused devices for every virtual machine. While our system is similar to previous work that suggests the same changes on a per-process level, the specific challenges of our environment require additional techniques to find good allocation strategies. In particular, we find that there are different types of memory that are used by a virtual machine and propose different strategies to allocate each type of memory. In addition, virtual machines are generally long-lived processes with a comparably large, mostly static memory footprint. We propose additional techniques to handle this type of behavior.

As with previous work in this area, we were unable to obtain hardware with working memory power-management capabilities. While the basic functionality is there, details in the implementation of the memory system prevent its use. Specifically, the use of memory interleaving prevents the use of memory power management. To evaluate the possible energy savings, we were restricted to trace driven analysis. This only

provides a lower-bound for the energy savings that can be obtained with real hardware.

This thesis is organized as follows. In the next chapter we will provide background on virtualization technology with a special focus on memory management. We then review previous work in the area of memory power-management that serves as the basis for our work. In Chapter 3 we describe our memory management system by expanding on solutions previously described and adapting them to our specific environment. We also explore new challenges and opportunities for energy savings and provide solutions for them. At the end of Chapter 3 we also give an outlook to future work based on our design. We then describe our prototypical implementation of parts of the proposed system on top of an existing hypervisor. In Chapter 4 we describe the required changes in detail and solve problems that arise from the environment. Using such a modified hypervisor, we then evaluate our changes in terms of performance and energy savings. In Chapter 6, we conclude with a summary of our system and qualification of the outcome of our experimentation.

Chapter 2

Background and Related Work

In this chapter we provide background on virtualization to help to understand the environment of this work, we also discuss the basics of memory hardware, and review previous work in the area.

In Section 2.1 we provide a short overview of complete machine virtualization and the basic techniques with an emphasis on the memory management process in the following sections. We focus on hardware assisted memory translation as this is the basis for our experimentation. This relatively new technology provides great performance improvements and makes virtualization performance almost en-par with real hardware. We also define some terms that are used throughout the remainder of this document.

We summarize the basics of memory hardware and power management for the memory subsystem in Section 2.2.

Finally, in Section 2.3, we then review related work in the field of power management of the memory subsystem. The review introduces a number of basic mechanisms and policies that are later referenced and improved upon in Chapter 3 as we describe our design.

2.1 Full Machine Emulation and Virtualization Basics

This thesis is concerned with full machine virtualization. A *Host System* provides one or more *Guest Systems* with the illusion that they are running on dedicated hardware. A guest system consists of a virtual processor, virtual main memory and a collection of virtual devices. The host system is a specialized software component that runs on the real hardware. This software is either an *Emulator* or a *Virtual Machine Hypervisor*. While an emulator—as the name suggests—emulates every single machine instruction, a hypervisor tries to run as much of the guest's code on the real hardware. The processor intercepts certain instructions and the hypervisor then emulates these instructions in order to provide the virtualization. In either case, memory that is provided to the guest needs to be virtualized.

The task of providing virtual main memory is similar to providing virtual memory in any multi-user operating system. Instead of a virtual address space for every process, there is a virtual main memory for every guest. As the memory system is the focus of this work, we will describe this in more detail in the following sections. To virtualize the processor the hypervisor uses the same mechanism as is used for a task switch in a traditional operating system: The active guest's processor state is saved, the

next guest's processor state is restored, and the new guest continues. The hypervisor intercepts and emulates all instructions that are used to program the device hardware, thus virtualizing the other devices that are required for full machine virtualization.

2.1.1 Virtual Main Memory

Virtual main memory is a special address space. The guest sees its virtual main memory as if it was real hardware memory. The guest can address the virtual main memory using a linear address. The hypervisor is responsible to provide the guest with this memory and must setup a translation that maps any valid guest address to real memory on the host. In this section we explain how this translation works and how the hypervisor creates the mapping.

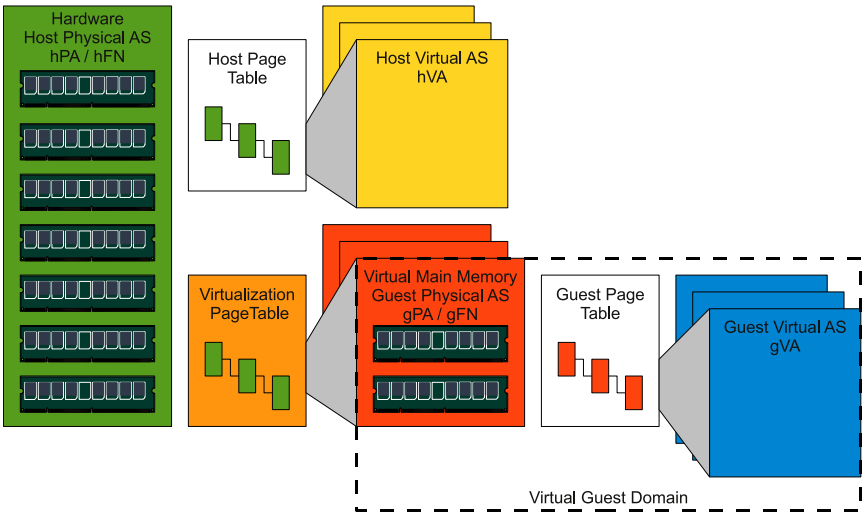


Figure 2.1: Different Address Spaces in Use With Full Machine Virtualization

Figure 2.1 shows the different address spaces that are used in our scenario. At the lowest level we have the host's hardware. A number of memory devices provide a linear physical address space that is addressed using the physical host frame number (hFN). The hypervisor provides virtual address spaces to its processes where they can manage their memory. This virtualization is provided by the host page tables that translate a host virtual address (hVA) to a host physical address (hPA)—similar to traditional operating systems. To provide the virtual main memory, the hypervisor uses a special kind of page table that translate a guest physical address (gPA)—inside the virtual main memory—to a host physical address (hPA). Inside a virtualization domain, the guest sees the virtual main memory as real hardware. The guest operating system may also use page tables to provide virtual address spaces. The guest page tables inside the virtualization translate a guest virtual address (gVA) to a guest physical address (gPA).

Virtualization page tables are currently available as a processor feature from both Intel and AMD in their latest processors [1, 16]. The feature is called "Enhanced Page Tables" or "Nested Page Tables", respectively. The basic mode of operation is equivalent in both implementations, but there are some differences. In the remainder of this document we will call this feature *Virtualization Page Table (VPT)* to refer to the general concept.

Where a VPT feature is not available, the hypervisor can use a concept known as *Shadow Page Tables*. The hypervisor constructs a specialized virtual address space in the host that is used while the guest is running. This address space translates directly from a guest virtual address to a host physical address. Keeping the shadow page tables synchronized with the guest is an expensive operation. As a result, memory throughput is increased up to 20-fold by using virtualization page tables for certain scenarios. Our design is based on VPTs, but could be made to work with shadow page tables as well.

Classic Host Page Table Lookup

In order to explain how the virtualization page tables work, we first take a look at normal page tables. Figure 2.2 illustrates the translation process. We assume here—and throughout the rest of this section— x86_64 hardware with 48 bit virtual address space. The objective is to translate from a 48 bit virtual host address to a host physical address. The input for this translation is the hVA in combination with the processor's `cr3` register that selects a page table hierarchy. First, the translation hardware locates the physical page that holds the root node for the active page table (PML4). The `cr3` register contains the hPA of that page. Once the hardware has located the root page directory it takes bit 47 through 39 to get a 9 bit index into the directory. Nine index bits mean there are $2^9 = 512$ entries in the first level, each holding one 64 bit entry. The index selects one of these entries that then holds the hPA for the second page directory level (PDP). The next 9 bit in the hVA (38 - 30) are used by the translation as an index into the page directory page, again resulting in a 64 bit entry that holds the hPA for the next level (PD). The same process continues, this time using bits 29 through 21 and resulting in the final mapping level, the page table. The index, obtained from bit 20 to 12, points to the final page table entry (PTE). The PTE again is a 64 bit value that holds the hPA for the final data page. The remaining 12 bit of the virtual address is a byte granularity index into the data page—addressing the 4,096 bytes in a page. The result of the translation is the hPA extracted from the PTE plus the offset.

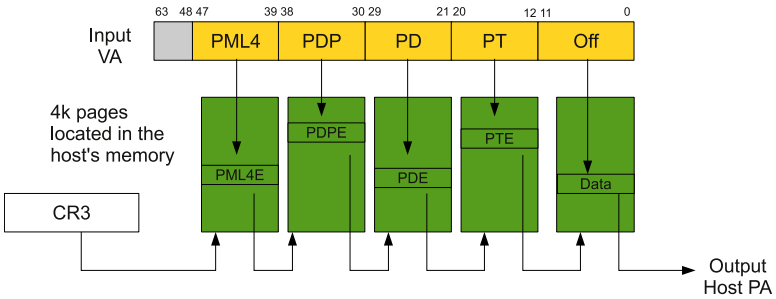


Figure 2.2: Page Table Processing Host AS

Every page directory and page table entry (P*E[1]) contains, apart from the hPA for the next level, some spare bits that are used to store access rights and a *reference* bit. At any of the intermediate levels there can also be a *non-present* mapping—a special value of the 64 bit entry. Such a mapping, as well as insufficient access rights, cause a

[1]We refer to the entries in the page table hierarchy—PML4E, PDPE, PDE and PTE—as P*E when the mapping level is not of interest.

page fault that the hypervisor must react to. We outline the process to resolve a page fault for virtualization page tables in a separate section below.

The hardware sets the referenced bit in the P*E that it encounters during the lookup. The referenced bit can be used to obtain information on which pages have been referenced recently. Traditional operating systems use this bit to implement *Least-Recently Used (LRU)* page reclaim. Our design also has to determine which pages have been referenced and will make use of the referenced bit (in the virtualization page tables) to do so.

Virtualization Page Table Lookup

In this section we compare the translation process for basic page tables described above to the virtualization page table processing depicted in Figure 2.3. The upper half is the guest page table that works exactly as in the host. All addresses in this structure are guest physical addresses (gPA), however. Thus, in order to find the underlying host physical address (hPA), an additional level of translation is required. This additional translation process is provided by the virtualization page table shown in the lower part of the figure. The figure makes clear why AMD calls their VPTs "nested page tables" [1]. For every translation step in the guest, there is a *nested* lookup in the VPT to find the underlying host page.

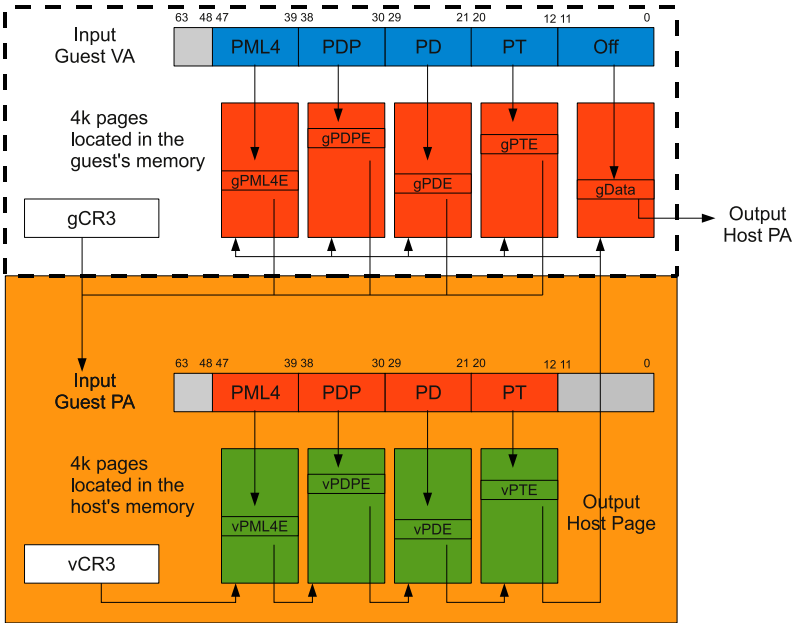


Figure 2.3: Virtualization Page Table Processing

The objective of the lookup procedure as a whole is to translate the guest virtual address—given as input at the top—to a host physical address so that the processor can fetch the data.

The translation process works similar to that of the traditional page table lookup above. The processor starts with the guest's `gCR3` register that selects the active virtual address space inside the guest. It contains the gPA for the root node (gPML4). To

continue the lookup, the processor needs to locate the physical host page that contains the data for this portion of the virtual main memory.

To find the host page that holds the gPML4, the processor now does the first nested lookup using the VPT. The lookup in the VPT itself also works exactly like the lookup in a normal page table. The input for the lookup is the gPA from the top and the contents of the `vCR3` register. The `vCR3` has the same function as the normal `cr3` register, but—instead of selecting a virtual address space—it selects the virtual main memory for the active guest. The processor walks the VPT to locate the vPTE that corresponds to the gPA given as input to the process. This vPTE contains the hPA of the page that contains the gPA in the input. With this information, the processor can continue the lookup in the guest page table.

As before, the processor takes a part of the gVA as an index into the current page directory or page table to arrive at a gP*E. This entry again contains a gPA and another nested lookup is required to continue with the translation.

The process continues in a similar fashion: The gPML4E is translated to the hPA of the gPDP using a nested lookup, the gPDPE to gPD, the gPDE to gPT, and finally the gPTE to the hPA of the host page that holds the actual data for the gVA. At this point the lookup is complete and the processor only has to add the 12 offset bits from the gVA in the input to the hPA from that last nested lookup to arrive at the final location.

Again, at any point of either page table, there can be a non-present mapping or a mapping with insufficient access rights. If such a mapping is encountered in the upper half of the lookup—the guest page table—the processor issues an exception in the guest operating system, which is responsible for setting up and maintaining this page table. Missing mappings in the VPT, on the other hand, cause an exception in the hypervisor, which is responsible for the VPT setup and maintenance. The process that is used by the hypervisor to resolve a non-present page fault in the VPT is described in the next section.

Performing the nested lookup is an expensive operation. For every step the processor has to read five memory locations: vPML4E, vPDPE, vPDE, vPTE and the gP*E. Thus, the implementations make use of a *Virtualization Translation Lookaside Buffer (vTLB)*. The vTLB stores intermediate translations from gPA to hPA. Instead of doing the costly VPT walk, the processor checks the TLB for a cached translation and uses it instead. In addition, there is the normal TLB that caches complete (g)VA to hPA translations that are used for virtualization and traditional lookups alike.

We assume it is because of this heavy use of buffering, that Intel's enhanced page tables do not have a referenced bit as normal page tables do. In order to maintain a referenced bit on the vP*Es, the implementation has to update the entries in memory whenever the corresponding vTLB entry is evicted from the buffer. This can be expensive. Unfortunately, this shortcoming in the design of EPTs makes it more difficult to figure out which vP*Es have been recently used. We provide a workaround for this problem in our software implementation, but a hardware solution would be beneficial not only for our work. The AMD implementation does provide a working referenced bit.

Constructing Virtual Page Tables and Virtual Main Memory Management

For our work, the most interesting part of the memory management is how the hypervisor creates the mappings for the virtual main memory. In this sections we describe how this is done. Figure 2.4 gives an overview of the process. The upper half shows the the translation from a guest virtual address space to the host main memory, which we just

discussed in detail. The lower part shows how the hypervisor creates a mapping. This process in used by the hypervisor whenever the hardware encounters a non-present mapping in the VPT.

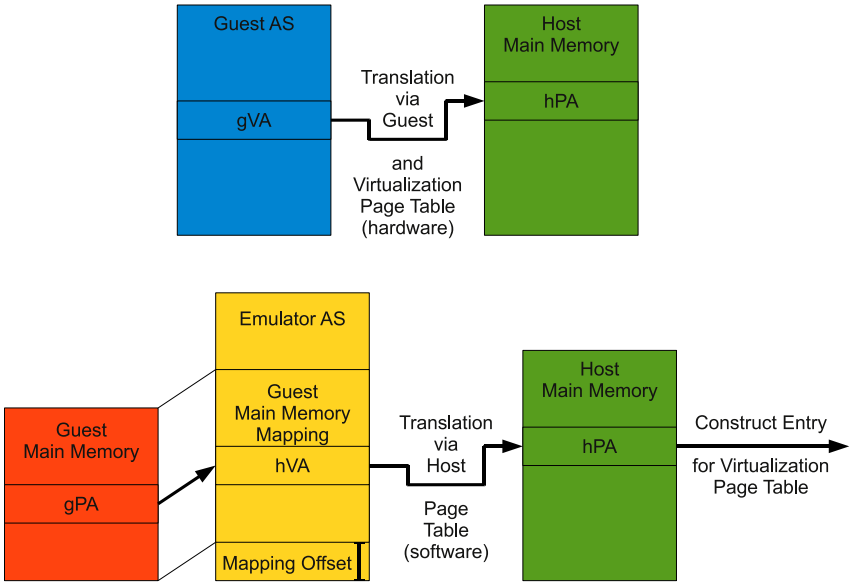


Figure 2.4: Address Translation and Virtualization Page Table Construction

In order to manage a guest's virtual main memory, the hypervisor creates a virtual host *Address Space (AS)*—called "Emulator AS" in the figure. This is a traditional virtual address space managed using a host page table. In this address space the hypervisor has a mapping for the guest's virtual main memory. The figure shows how the virtual main memory is embedded in the emulator AS. With this setup, the hypervisor can easily calculate the hVA inside the emulator AS that corresponds to any valid gPA. The figure shows the virtual main memory as mapped contiguously inside the address space. In order to transform a gPA to a hVA the hypervisor only needs to add the mapping offset. In praxis, the mapping is a bit more complex than this as the virtual main memory can be split up into chunks for better management, but the principle remains the same.

To construct an entry in the VPT, the hypervisor needs the hPA—not a hVA inside the emulator AS—that corresponds to the gPA. In order to find this information, the hypervisor consults the host page table that is used to manage the emulator AS. The hypervisor walks this mapping structure in software in the same way the hardware would (see above for details), using the hVA obtained before as the input for the lookup. The result of the walk is a PTE at the last level of the emulator AS page table or the information that this portion of the AS has not been mapped yet, as the walk encounters a non-present mapping.

In case the walk encounters a "non-present" mapping, the hypervisor first has to allocate memory to back this mapping and modify the emulator AS page table accordingly. This is the crucial point in the process for our work. When the hypervisor does this allocation it decides the physical location for a section of the virtual main memory. We describe a policy to guide this process as the core of our work.

Once the allocation succeeded or if the mapping was already valid, the hypervisor

knows the hPA that corresponds to the gPA. The hPA is either the location of the new allocation or the hypervisor can extract this information from the PTE that resulted of the lookup. At this point the hypervisor can enter the found mapping between gPA and hPA into the virtualization page table.

Entering a mapping into a page table hierarchy works similar to a lookup. The gPA, for which the mapping should be constructed, is used as input. The tables are walked in software. If a non-present mapping is encountered at any vP*E on the way, the process has to allocate a new page to serve as page directory[2] or page table in that spot. With the newly allocated page the lookup process can fix the non-present vP*E to point to the new page directory or table and continue. The result of the lookup is the vPTE that maps the gPA. The hypervisor modifies the vPTE to point to the hPA that it has found above and the mapping is established.

This whole process is expensive. In order to construct a mapping for a single page in the virtual main memory the hypervisor has to walk two page table hierarchies in software—resulting in eight memory accesses. In addition, constructing the new mapping in the VPT can require a number of allocations and memory writes. This effort is set off by the fact that any future access by the guest to that part of the virtual main memory is completely handled in hardware with no intervention by the hypervisor.

2.2 Power-Management for the Memory Subsystem

In this section we provide an overview of the memory system and the technology used, in order to explain how we can save energy in the memory subsystem. As we mentioned above, the main memory of a computer is provided by a number of memory devices. The dominant technology for *Random-Access Memory (RAM)* today is usually a form of *Double Data Rate Synchronous DRAM (DDR-SDRAM)*. This technology allows the memory controller to put a memory device—as a whole—into a lower power mode while no data from the module is required. In this regard, DDR-SDRAM is different from the *Rambus DRAM (RDRAM)* technology. RDRAM allows for a more fine-grained power-management that enables the controller to select a different power state for smaller parts of a memory device. The majority of previous work [8, 12] in software driven memory power-management was conducted using the RDRAM technology for evaluation. We show during our design and evaluation that, while smaller management units can help to provide better results, our target can already benefit from the more coarse-grained management present in DDR-SDRAM. This is a significant finding, as the RDRAM technology has since been discontinued [10].

Size (MB)	tRFC (ns)	IDD2N	IDD2P0	IDD3N	IDD3P	IDD4R/W	IDD5B	IDD6
		(mA)						
512	X [a]	220	44	240	180	<940	840	40
1024	110	440	88	480	360	<1520	1680	80
2048	160	990	200	1035	810	<2205	2385	180

[a] The specification does not provide a value for tRFC for 512MB modules, but we assume that it will be somewhere between 60 and 100 ns based on the values for bigger modules.

Table 2.1: DRAM Specifications

During our evaluation (see Chapter 5) we use DDR3-SDRAM. The memory con-

[2] We use “page directory” as a synonym for any page that is not the final “page table”

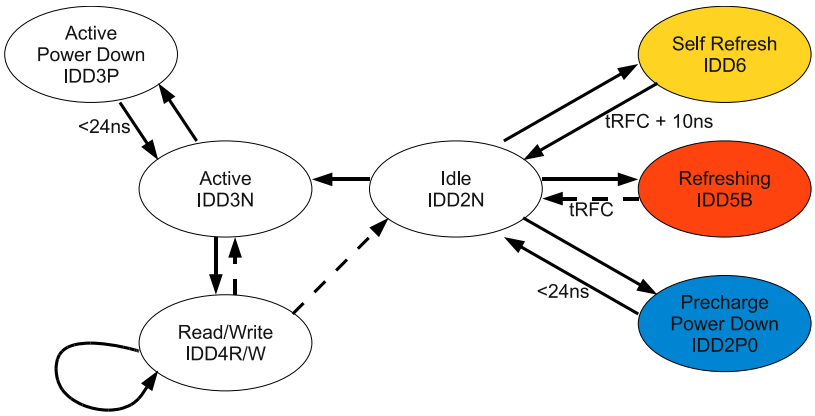


Figure 2.5: DRAM Power States

troller can put a DDR3-SDRAM device into a number of different states. A simplified state machine can be found in Figure 2.5. Each state has a different power-consumption, referenced by symbolic names in the figure. Table 2.1 provides the actual values for these symbols based on the specifications [19] of the memory modules used during the evaluation. In addition, it takes a certain amount of time to change the state of a memory device. Significant times are shown in the figure next to the state transition. Dotted lines denote automatic state transitions after a timeout or completion of a task. While putting a device into a low-power mode (Self Refresh, Precharge/Active Power Down) is almost instantaneous, returning the device into active mode does take some time as certain components of the memory device are resynchronized with the memory bus clock.

*Dynamic*RAM stores the information by charging a capacitor. In order to retain the information, this charge has to be refreshed on a regular basis. This is the purpose of the *Refreshing* state on the right hand side of the figure. The memory controller is responsible to enter this state regularly. This is also true while the device is in either *Active* or *Precharge Power Down* mode. While these modes can be used for saving energy and provide a relative short resynchronization time, the *Self Refresh* mode is the most interesting in terms of saving energy. While a device is in self refresh, it is the responsibility of the device to take care of refreshing the capacitors. Using a slow and steady refresh mechanism, self refresh is much more effective than the bulk refresh that is done during a normal refresh issued by the memory controller. Our work focuses on the *Self Refresh* mode for energy reduction. We show in Chapter 3 that the delay that is introduced when the device exits self refresh can be hidden in other processing overhead and thus does not impact on performance in a negative way.

2.3 Related Work

The basic idea for our thesis is to change the policy for memory allocations in a way that allows to save energy in the memory system. In 2000, Lebeck et al [12] first introduce *Power Aware Page Allocations*. Their work is the basis for most of the work done in the context of memory power management, since. The main contribution is the idea to allocate memory from as few memory devices as possible, in order to allow the hardware to power down the unused devices. The allocation policy used is *Sequential*

First Touch. The concept behind this policy is to fill a single device before moving to the next. The concept of power aware page allocations is further refined by a number of papers [8, 13, 18].

A 2003 paper by Huang et al [8] takes the basic concept presented in [12] and adds two new concepts: Sequential first touch on a per-process basis and the use of *Migration* to optimize the initial memory placement. In the Lebeck paper [12], sequential first touch is used on a global scale. All memory allocations done by the operating system are satisfied from the first node until this node runs out of memory and subsequent allocations are then placed on the second node, and so forth. In contrast, the per-process policy considers individual processes in an operating system. The allocation policy assigns an *active node set* to each process and uses sequential first touch to minimize this set for every process. Now the hardware can switch off all nodes that are not part of the active node set of the currently running process. In addition, memory used by a process outside of the active node set—due to, for example, shared libraries—is migrated to nodes inside the active node set.

In 2007, Lee and others [13] point out a major shortcoming in the approach of the Huang paper [8] in that it does not consider the kernel memory and—in particular—the buffer cache used by processes. This oversight results in unnecessary memory device wake-ups, energy waste and performance degradation. The solution is to allocate this memory in a similar fashion. The operating system has the knowledge which process is responsible for a buffer cache allocation and can use the active node set of this process to pick a node. While these findings are not directly applicable to our work, we consider different kinds of memory used by a guest in our design. We also discuss the implications of guest access to secondary storage, which is a similar problem as the buffer cache. Due to the different environment for our work, we recommend to allocate buffer memory from a central *system node* instead.

In the same category are findings by Pandey et al [18] that are concerned with *Direct Memory Access (DMA)*—done by device hardware—the interactions with the memory subsystem, and their impact on power management. In a virtualized environment there are three different types of DMA. As a traditional operating system, the hypervisor contains device drivers that can issue DMA operations in order to send/receive data to/from a hardware device. In addition, the guest operating system can use virtual device DMA to request data from the hypervisor. The result of virtual DMA operations is a memory copy between host and guest. Alternatively, the host can use page sharing. Finally, when the hypervisor grants a guest exclusive access to a device, the guest can issue real DMA operations to that device [27]. These operations are translated by a specialized *Memory Management Unit* called *IOMMU* in order to maintain the virtualization. Our design does not directly consider device access, but we give some guidance for future work in Chapter 3.

In addition to the refinements of the original power aware page allocation, several improvements to the memory hardware were suggested [3, 5, 9] to help with implementing and fully realizing the potential of power aware allocation techniques. As with other targets for power management (cf. disk power management [21]), using heuristics in the hardware for the basis of the power-management decisions provides only sub-optimal results. The hardware has less information than the operating system to base its decision on. The memory controller is mostly unaware of the concept of a process as the principle of memory accesses, and the scheduling decisions performed by the operating system. In addition, the hardware can only react to the placement of memory allocations that is specified by the operating system. To overcome this problem co-operative techniques—where the OS is able to provide hints to the hardware—

have been analyzed [3, 9]. The work shows that significant improvements are possible with specialized hardware. The interface to such hardware—as designed in the cited articles—is based on the basic principle of the previously discussed work, which concentrates on the software alone. As our design is also based on these same principles, it will be possible to use our work in conjunction with the specialized hardware, as well.

In contrast to the work discussed so far, our design does not only consider the memory allocated to a process/guest, but also the *Working Set* of a guest. This is necessary as a single guest allocates a large amount of memory of which only a part is used during normal operation. There are a number of mechanisms to obtain information about the working set of a process/guest or the operating system as a whole. Delaluz et al [4] use working set detection in the operating system for memory power management. To detect the working set, they use repeated page faults and the page table reference bits. Our design uses page faults in the VPT in a similar fashion. But in order to avoid the performance penalty from repeated page faults, we uses an idea presented in an article from 2004 by Zhou et al [28]. The objective of their work is to track the *page miss ratio curve*, which is not directly applicable to our work. Their approach, however, reflects in our design of the working set detection in that we balance between working set overestimation and cost of the analysis.

To our knowledge, there is no previous work that investigates memory power management for virtual machines. In this thesis we take the findings presented in previous work [8, 12, 13, 18]—discussed above—and adapt them to virtual machine hypervisors for the first time.

Chapter 3

Design

In this chapter we describe our design for a power-aware memory management system for virtual machines. We take techniques from the reviewed articles in the previous chapter, which are mostly based on processes in traditional operating systems, and adapt these techniques to our environment. We identify a number of differences between traditional processes and virtual machines: Virtual machines have a large memory footprint that is known upon startup, where traditional processes have varying footprints that are unknown on startup and can change dramatically during the runtime of a process. Compared to traditional processes, virtual machines are consistently long-lived. Our design also considers changes in memory technology since previous work has been conducted.

3.1 Basic Architecture

The system consists of a host system, made up of a processor, a set of devices and memory. On the host system we have one or more virtual guest systems that also have their own *virtual* processor, devices and main memory. The host processor is multiplexed in time. Each VM receives a share of the host processor time to run its calculations. The devices are either also multiplexed in time (e.g. a network card) or they are multiplexed in space (e.g. hard disks). In addition a single host device can be given to a guest exclusively (e.g. an USB port and connected printer, or a serial line).

The focus of our work is the main memory. Multiplexing this resource in time is disadvantageous as it would mean that we have to save and restore the virtual main memory contents of each individual guest when switching between them. The logical consequence is to multiplex the main memory in space.

The architecture and multiplexing techniques are illustrated in Figure 3.1. As described in Chapter 2, we work in an environment where the host's memory is comprised of a set of memory nodes. Each node holds a certain range of the host's main memory and each node can be switched to an individual power mode. If we—as hinted at in the figure—assign a separate memory node to each guest VM, we can switch off the other nodes while this guest is running and—as a result—save energy. In Figure 3.1, we could switch off the red and yellow memory nodes while the blue VM is running and vice versa.

The goal for our changes to the memory management is to reduce the static energy dissipation in the memory system. Our design considers a system that can run only one

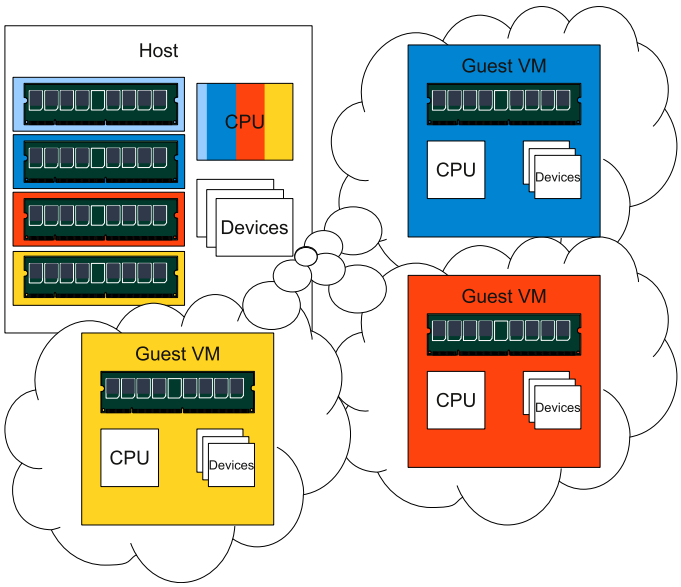


Figure 3.1: System overview showing a host and three guest VMs with associated processors, devices and memory

process or—as in our case—one virtual machine at a time. In such a system reducing the static energy dissipation is achieved by reducing the number of memory nodes that are in use by each virtual machine. As a result, the scheduler can put all unused memory nodes into a lower power mode when switching between VMs and thus save energy.

In this chapter, we introduce and describe two basic techniques that help to ensure that a minimal number of memory nodes are used by each VM.

First we describe a static allocation policy that decides from which memory node a memory request on behalf of a certain VM should be satisfied. The allocation policy is based on recommendations from the referenced articles in Section 2.3. We adapt the *Sequential First Touch* Policy and—in Section 3.4.2 below—introduce a variant with reservations that is specific to our environment.

In addition to the initial static allocation policy, our system gathers dynamic information about a VM's access patterns. We introduce and describe techniques to obtain this information based on the *virtualization page table* used to map the virtual main memory of a guest. Based on the access patterns we determine the *working set* of a guest in Section 3.3.1. In Section 3.3.2 we describe a policy that improves the initial placement of a VM's virtual main memory using the information about the working set. Based on this policy our design *migrates* pages between nodes in order to co-locate the pages in the working set on fewer memory nodes.

In Section 3.4, we expand the focus of our design and identify additional opportunities to save energy by considering all active guests and interactions between them. We show that the scheduling sequence of individual VMs can impact the energy consumption and outline strategies to optimize the scheduling policy based on the working set of individual guest. Section 3.4.3 describes a novel approach to handle the reduced working set of idle VMs. We show that additional energy can be saved by duplicating the idle working set. Finally, we discuss the impact of *sharing pages* between different

guests, in Section 3.4.4. We demonstrate that sharing pages—in general—is adverse to our approach on saving energy.

Finally, in Section 3.5, we discuss the implications of our design on environments other than the primary focus of our work. Section 3.5.1 describes *para-virtualization* techniques that can be helpful to our work. The primary focus of this work is pure virtualization without modifications to the guest. In Section 3.5.2 we revisit the decision to focus on single processor systems and describe what is necessary to expand our design to work with *multi processor* and *Non-uniform Memory Access (NUMA)* systems.

3.2 Reducing Energy for a Single VM

To reduce the static energy used by a single guest, we want to use a few memory nodes as possible to store the memory used by the guest. There are two ways we can assure that: First, we design an allocation policy that ensures that all memory allocations on behalf of the guest end up on as few nodes as possible. Our allocation policy is the same as proposed in previous work [8, 12]—sequential first touch. The second possible way to ensure that the guest uses as few memory nodes as possible, is a dynamic approach. We determine the working set of a guest during runtime and use migration to co-locate the working set on fewer nodes.

3.2.1 Allocation Policy

We use the *Sequential First Touch* allocation policy, described in the referenced articles [8, 12, 13]. For every guest, we keep a list of *active* memory nodes. This list contains all memory nodes from which the guest has previously allocated memory. For a new allocation, we prefer memory from a node in this list over adding a new node. For the first allocation and when no more memory is available in any node in the list, the policy picks the memory node with the most free memory and adds it to the list. This is to ensure that as many future allocations as possible can be satisfied from the newly added node.

This simple, greedy policy tries to ensure that the memory for each guest spans a minimum number of nodes. In addition, the policy results in a placement where different guest occupy disjunct active node sets, under the precondition that initially all nodes are empty and of equal size. Current technology usually provides memory nodes of equal size. In this situation, the allocation policy uses the first empty node for the first guest. As a result, the available memory in this node is reduced. The first allocation from the second guest is then serviced from a different, empty node. The allocation policy will service following requests for memory from the first guest from the first node and requests from the second guest from the second node. The process continues in a similar fashion until there is at least one allocation in each memory node. At this point, the allocation policy will select the node with the least allocations for a new guest. This means that the new guest shares a node with a guest that did not allocate a lot of memory so far. The hope is that neither guest will allocate much more memory in the future and their allocations can be satisfied from that same node.

With the allocation policy, our design can already save energy. The active node list contains all memory nodes that need to be active for a given guest. As a result, all the remaining nodes in the system do not have to be active while this guest is running. Our design uses the active node list when the hypervisor switches between guests. We put

all memory nodes in the active node set of the new guest into the active state, and all remaining nodes into a low power state.

In the next section, we take a closer look at the memory that is used by a guest. We identify different types of memory and describe how we apply our allocation policy to these individual portions of memory.

3.2.2 Different Types of Memory

In our environment of full-machine virtualization there are different types of memory that "belong" to a VM. Independent of the particular virtualization technology the memory for a VM is comprised of:

1. Virtual main memory
2. Emulator and hypervisor code and static data
3. Dynamic state and data associated with a VM
4. Memory management data structures—the virtualization page table hierarchy

This is illustrated in Figure 3.2. The figure shows two individual VMs, sharing code and data, their own dynamic data, and virtual main memory. The virtual main memory also includes the page tables inside the VM. In the hypervisor there is also static code and data, a state for each VM, and the virtualization page tables. Based on the different properties of each of these memory regions, our design will treat each type of memory differently to obtain the best results. Previous work focusing on per-process energy savings provides some guidance [8].

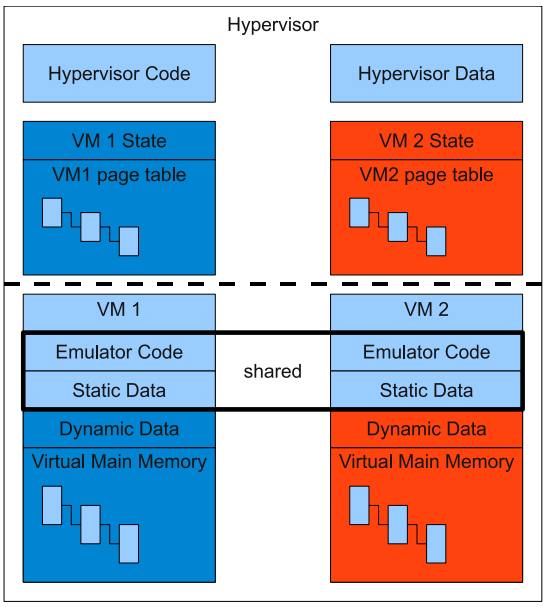


Figure 3.2: Different Types of Memory in the VM System

To allow sharing of the static memory regions, our design will keep the static code and data of the emulator and hypervisor in a single node that is shared by all VMs. This

is analogous to the placement of shared libraries and kernel code in previous work. For example, the Huang paper [8] explicitly keeps all the kernel code and data in a central system node and allocates shared libraries from a special node set, which is shared by all processes that use those libraries. The special function of the hypervisor code as the central point for the memory management and all scheduling decisions is an additional reason that these memory regions need to be on a central node that stays active. The alternative—to place a copy of this data on each VM's active set—is not only wasteful in space, but also has adverse impact on performance. Previous work [8] shows that the wakeup delay required to switch a memory node from a low energy state back to standby can be hidden in the task switch. Their idea is to switch the power state of a memory device as soon as the scheduling decision has been made. The delay that is required to re-enable the node is then hidden behind the additional processing that is required to execute the actual task switch. This approach only works if all the code and data, which are required to perform the scheduling decision and the task switch, are located on a central node that stays active. It should be noted that switching between virtual machines, in general, is more expensive than switching between normal processes—additional registers need to be saved and restored and programming the virtualized `cr3` register is an expensive operation. We conclude that our implementation can easily hide the wakeup delay in the task switch processing, based on the numbers given in the referenced work.

In addition we need all the data that is required for the scheduling decision and task switch to be on the central node as well. This accounts for most of the dynamic state and data that is allocated by the hypervisor to manage a VM. This data is mostly the state of the virtual CPU that needs to be copied to and from the real CPU when switching between VMs, as well as bookkeeping and accounting. It should also be noted that this is a comparable small amount of data.

For the virtualization page tables we decided to also place them on the central system node. This decision was made for a number of reasons: Eventhough this data structure is only used while the associated VM is running, the migration policy, which we describe in Section 3.3.2, also accesses the VPT. The migration decisions are made from a background task. If we place the VPT data on separate nodes for each VM, the migration policy would have to enable these nodes to perform its duties. Doing so would waste energy. The observation that it is customary to provide VMs with a "normal"—power of two sized—amount of virtual memory, provides us with another reason to keep the VPT data on the system node. As hardware main memory is usually power of two sized, off-the-shelf operating systems are optimized to work under this precondition. For the same reason, the memory power management units are power of two sized, as well. The result of all these sizes lining up, is that the virtual main memory of a single VM will just fit into a number of memory nodes. For example, 4GB of virtual main memory in a system with 512MB memory nodes fills up 8 nodes. If we were to add the VPT data to the same nodes as the virtual main memory, the nodes would overflow by the size of the VPT data and lead to fragmentation. The final reason to place the VPT data in the system node is the fact that previous work [8] implicitly keeps the page tables, which—in their environment—serve the same purpose as the VPT in our environment, on a central node as well.

Finally, the only type of memory is the virtual main memory itself. In order to apply the allocation policy to any memory at all, we will apply it to the virtual main memory. In addition, the virtual main memory is the only type of memory that is exclusively used by a single VM and not required by the hypervisor during the task switch. The migration task (cf. Section 3.3.2) does need access to the virtual main

memory to migrate the data, but does not need access to make the decision. If memory sharing between VMs is enabled, we need special placement of the shared regions as we describe in Section 3.4.4, but this is orthogonal to the initial placement performed by the allocation policy.

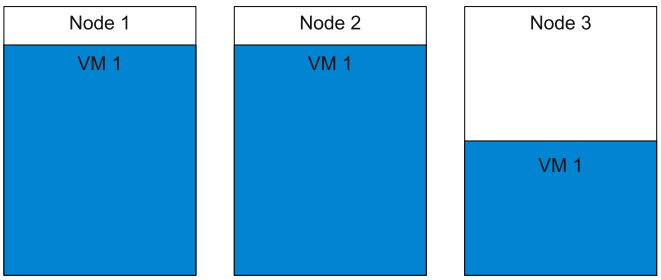
3.3 Improvements for a Single VM

The allocation policy described herein above already saves energy compared to the default buddy system [11] based allocation policy, that places an allocation on an, in-effect, random node. However, as soon as a single VM has an active node set of more than one node there is room for optimization. We have to consider the current hardware trends to explain why that is. Since the RAMBUS technology, with fine-grained memory management capabilities, has not met consumer acceptance, DRAM memory power-management has become more coarse-grained in general. The current market does not provide node sizes smaller than 512MB. As a result a VM's virtual main memory will usually only span a few nodes. In addition, we assume that any meaningful computation does not keep all of a VM's virtual main memory active all the time. Instead there is a working set to consider. We define a VM's working set as the collection of pages in the VM's virtual main memory that this VM has touched recently. If we can keep the current working set on a single node out of the VM's active node set, we can further decrease the VM's power dissipation as we can switch the other nodes in the active node set to a lower power mode, as well. We call these nodes idle–active nodes, while the nodes that hold pages that are part of the working set are called active–active nodes.

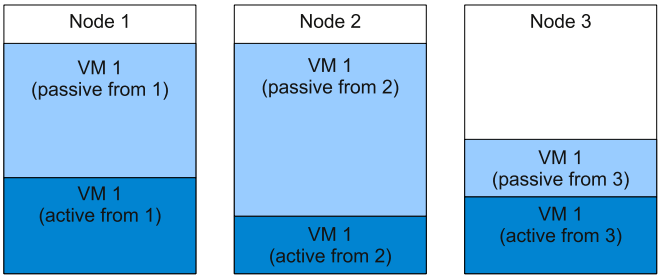
An example of the intended outcome of this section is shown in Figure 3.3. From the initial placement of a VM's virtual main memory shown in Figure 3.3(a) we would like to arrive at an improved placement shown in Figure 3.3(e). The first task to make this possible is to find a way to distinguish between active and passive portions of the memory in each node (c.f. Figure 3.3(b)). For that purpose we need to design a way to detect the active pages for a VM. In a second step, we then have to find a policy for the migration (c.f. Figure 3.3(c) and 3.3(d)).

3.3.1 Working Set Detection

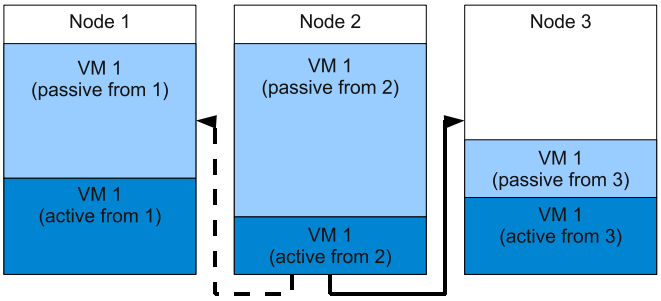
A VM's working set is the collection of pages that this VM used recently. There are several ways to obtain this information. The straight forward approach is to trace every memory access of the VM and keep an ordered list of all the pages, moving that with the most recent access to the front of the list. Based on such a list, the working set is easily obtained by taking the first N pages in that list. This, however, is very expensive. Instead we model our working set detection in the same way as traditional operating systems obtain information about unused pages in order to swap them out to secondary storage. We know all pages that are in use by a VM from looking at the virtualization page table. Every page that has a mapping in the table has been used by the VM at some point in time. If we allow the table to grow until the whole virtual memory is mapped, we lose the information which pages have been recently referenced. Instead we restrict the number of mappings that are allowed. When another mapping needs to be established and there are no more mappings allowed, we release the least-recently-used mappings in order to make room. With this simple approach the pages that are mapped in the virtualization page table directly match the working set.



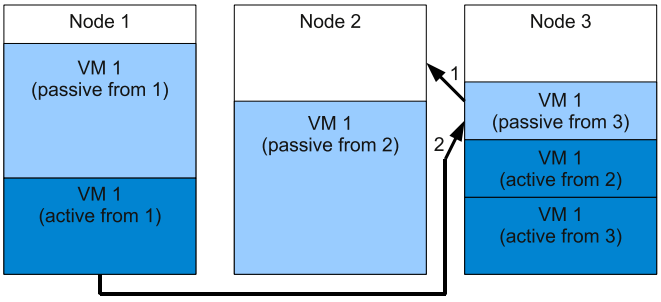
(a) VM with Three Active–Active Nodes



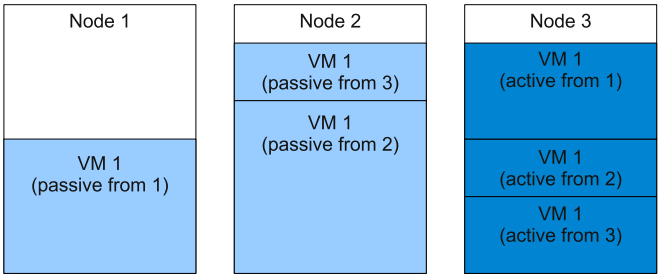
(b) Active and Passive Memory



(c) First Migration



(d) Second Migration—Making Room in the Target Node First



(e) Final Placement—One Active–Active Node

Figure 3.3: Migrating Active Memory

To avoid over-estimation of the working set, which would mean that we have to migrate too many pages later on, we want to keep the number of allowed mappings to a minimum. On the other hand, if we under-estimate the working set we will suffer a performance hit. When the guest references a larger number of pages than we allow to be mapped in a short amount of time, we will end up releasing and re-establishing the same mappings over and over. In addition we lose information about the age of a mapping as we have to throw out valid active mappings in order to accommodate new ones. To balance between over- and under-estimation, we need a way to guess the size of the actual working set at runtime. Our approach is to limit the rate at which we reclaim least-recently-used mappings. If this rate exceeds a threshold, we allow more mappings to be established instead of reclaiming an old mapping. As a result, when the guest references a lot of pages in a short time the number of mappings grows—and with it the size of the working set. In addition to this, we need a counter measure that reduces the size of the working set again after we detect that the guest no longer references many pages.

Once the page reclaim rate stays below the threshold for a certain amount of time, our design uses the page reclaim mechanism to find any mappings that have not been used recently. In order to reduce the size of the working set, we then release these mappings.

Now that we have a way to find all pages that are part of a guest's working set, we need to transform this information in a way that helps us with our migration decision. For this we need two things:

1. The number of active pages in each node.

2. The time each page has been active.

The number of active pages in each node is easy to obtain. We simply add an accounting mechanism to each place that adds or removes a mapping in the VPT. Tracking the time for each page individually is infeasible—both in space and time. The virtual main memory of a node consists of hundred thousand pages or more. Storing even a small timestamp information for each of them quickly grows to a considerable amount of data. In addition, the mechanism to add a mapping to the virtualization page table, which is the point where we can add the accounting, is optimized for performance and any additional processing has to be designed accordingly, in order to avoid performance loss. Furthermore, such detailed information would also overwhelm any policy decision based on that information, as the policy would have to consider every piece of data. Instead of tracking the age of an individual page, we track the age of each active–active node. By this we mean the time since the last page from that node has been added to the working set. This information is easily obtained and it provides us with enough insight to make a migration decision. Looking at a node's age, we know that all pages in that node have been active for at least that long. Once we consider the active pages on a node to be "old enough" we can try to move them to a "better node" in order to optimize the placement. The next section describes a policy to determine when a node is "old enough" and how to find a "better node".

3.3.2 Page Migration

In this section, we explain our design of a migration policy based on the information from working set detection. We outline how our policy selects a source and target node for a migration. The first step to make a migration decision is to find a node as the

source of the migration. The section “Break Even Time” below describes how our policy finds a candidate based on the working set detection. In the next section—entitled “Node Selection Policy”—we then show how our design picks one of the candidates for the source and how it finds a target node, as well.

Break Even Time

Looking at other work in power-management, we quickly conclude that “old enough” will somehow relate to the break-even time—the time it takes to set off the additional energy invested into the migration by the anticipated savings due to the smaller active–active node set. The premise is two-fold here. On one hand we except that a page that has been active for a long time will stay active. This means that we do not move the page to an active–active node just so that it is removed from the working set shortly thereafter. On the other hand we also hope that once a node has reached a certain age, no other pages from that node will be added to the working set in the immediate future. If this second part does not hold, we can not materialize the anticipated energy savings as the node does not stay idle–active long enough. We argue that the second part holds due to the fact that pages touched together for the first time—thus allocated from a single node at first—will also be touched together later on. The evaluation will show that this is true for some guest operating systems. Guest operating systems that exhibit more random access patterns will not benefit from migration as much.

We found two possible ways to address this problem. One way is to increase the time before we consider a node to be “old enough”, once we detect that an idle–active node has switched back to active–active too soon. This is in line with other power-management policies that use exponential back-off to increase the break-even time once they detect violation. Another approach is to force the node to stay idle. This can be done by moving pages on-demand. As the hypervisor receives a page fault that would add a page from a newly idle–active node to the working set, it switches that node back on, only to copy the requested page to another already active–active node. We got the idea for this from initial evaluation results, which show that after a successful migration we sometimes receive page faults for a few “late” pages from the old node. While increasing the wait time would also catch these “late” pages, we found that we can increase the savings by on-demand migration, as “late” pages happen rather seldom for most tested guest operating systems. Once we did migrate the late pages, the node usually stays idle for a long time. Switching early and the longer total idle time then sets off the additional energy that was payed for the on-demand migration. Our implementation of the on-fault migration has to keep statistics to ensure that there is a benefit from the mechanism. Otherwise we will switch to a back-off mechanism instead if runtime behavior demands.

To calculate a specific value for the break-even time we have to consider the particular hardware that we work with. The break-even time is the time before the additional effort—in our case the cost of the migration—is set off by the savings that result of the effort—in our case the energy saved by being able to put an additional node into a lower power mode. The anticipated energy savings can be easily determined by looking at the specifications of the DRAM module in question. The energy required for the migration is more complicated to calculate, however. We were unable to make meaningful measurements on our target platform. We also failed to find exemplary measurements to guide us. We therefore resort to worst case calculation base on the specifications of the memory modules [19] and benchmark results.

The DDR3-1066 modules we use clock in at about 1.6 GB/s for uncached, page

sized reads and writes. Or 2.4 ms to read or write a single page. Now we assume that the node stays in the highest power mode for the whole duration of the copy. With this we can determine an upper bound for the energy.

$$E_{copy} = (V_{DDQ} \times (I_{DD4R} + I_{DD4W} - 2 \times I_{DD2N})) \times 2.4 \times 10^{-6} s$$

Where V_{DDQ} is the supply voltage (1.5V), $I_{DD4R/W}$ the operating burst read/write current and I_{DD2N} the precharge standby current. The anticipated energy saving can be calculated by:

$$P_{save} = V_{DDQ} \times (I_{DD2N} - I_{DD6})$$

with I_{DD6} being the self refresh current. Thus the break-even time for a single page migration is:

$$t_{be} = E_{copy}/P_{save}$$

Size (MB)	I_{DD2N}	$I_{DD4R/W}$ (mA)	I_{DD6}	E_{copy} (mJ)	P_{save} (mW)	t_{be} (ms)
512	220	940	40	5.2	270	19.3
1024	440	1520	80	7.8	540	14.4
2048	990	2205	180	8.7	1215	7.2

Table 3.1: Break-Even Time for different Memory Module Sizes

Table 3.1 shows the actual numbers for different memory device sizes. For our implementation, we add another 40% to the calculated break-even time. This results in 27ms for 512MB nodes, 20ms for 1024MB nodes, and 10ms for 2048MB nodes. We deem 40% a realistic, conservative estimate for the additional energy required in the memory system.

With the break-even time, we can also further qualify when on-demand migration is beneficial. Due to the additional overhead for powering the idle–active node up and down for the migration we add a tenfold safety-margin and consider on-demand migration to be beneficial if it happens at a rate of below 1/100ms for 2048MB nodes, 1/200ms for 1024MB nodes, and 1/270ms for 512MB nodes.

Node Selection Policy

With the break-even time, we now have a criteria to determine when a node is old enough to be considered as the source node for a migration. The migration policy now has to determine how to select a target node for the pages that we would like to migrate off the “old node”. The first step for migration is to find an “old node” as the source for a possible migration. With the above definitions it is easy to define the criteria for a node to be considered: First we want to be sure that no page from the node has been added to the working set recently. Then we need to consider how many pages are active from that node in order to calculate the required time for the migration to pay off. The policy multiplies the number of active pages in the source node with the break-even time for a single page migration—as defined in the previous section—to arrive at the break-even time for migrating all active pages. Once the last time a page from the node has been added to the working set is longer ago then the calculated break-even

time, we consider the node as potential source for migration. Now we need to find a target node where we can migrate the active pages to. We want to choose a node from the VM's active–active node set. Preferably one that will stay active for a long time. In addition we have to find room in that node. To ensure that the target node will stay active we prefer the node with the most active pages. This also helps to avoid expensive migrations in the future. If there is enough room on the selected target node we can perform the migration. Otherwise we have three choices:

1. Select another active–active node as the target.
2. Make room in the target node by moving idle pages to another node.
3. Defer the migration.

Our policy tries theses options in the given order. First, we attempt to find an alternative target node in the active–active node set. Given our considerations (see Section 3.2.2) about typical node sizes as compared to typical virtual memory sizes, we conclude that it is unlikely that there are many potential target nodes. As we describe above, the virtual main memory of a single guest will only occupy a few nodes. In addition, as normal sizes for the virtual main memory align with the size of normal memory nodes, all used memory nodes will typically be fully populated.

If the policy can not find an alternative target node, it then tries the second option. Making room in the target node requires additional effort. We first have to move some pages away from the target node. This additional effort leads to the third option. The policy first calculates the total effort required for the migration. To obtain this, it multiplies the number of pages that it needs to move away from the target node with the break-even time to migrate a single page. The resulting time is then added to the break-even time to migrate the active pages from the source node. The policy then checks if the total time is greater than the age of the source node. If this is the case, the migration can take place, otherwise we choose the third option and defer the migration until the source node is old enough.

The example given in Figure 3.3 shows the two scenarios. The first migration in Figure 3.3(c) uses the first option. In the depicted case, node 1 is the first potential target as it has the most active pages. Since there is no room left in node 1, however, the policy checks for an alternative target. We find that node 2 is also part of the active–active node set and has enough room to accommodate the active pages from the source node.

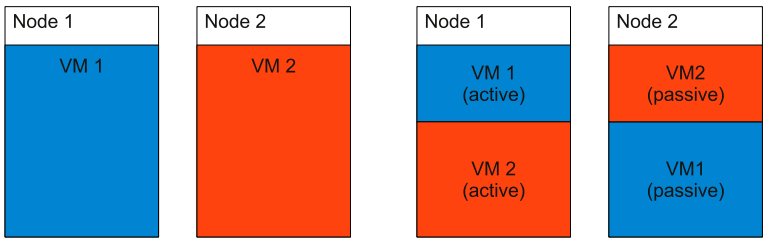
Figure 3.3(d) shows the second possible case. There is not enough room for the active pages in the target node—node 3. In addition, this is the only node with active pages, so the policy does not find an alternative target node. Instead, we first have to make room in the target before we can perform the original migration. The policy moves the idle pages in the target node to another node in the active node set of the VM. Finally, the original migration from node 1 to node 3 can take place.

3.4 Reducing Energy for All VMs

Up to this point, our design focuses on a single VM and how to minimize the energy dissipation in the memory system of that one VM. In this section we take a step back and look at the whole system in order to identify additional opportunities for saving energy.

3.4.1 Inter-VM Migration

As we mentioned above, the sequential first touch allocation policy places different VMs on disjunct node sets until it runs out of empty nodes. While this is beneficial at first—as it ensures that as many pages of a single VM end up in the same node—it is not optimal on the global scale. Consider the two placements given in Figure 3.4: In the static case each VM is assigned a separate node. While VM 1 is running, node 1 is active and node 2 can be put into a lower energy mode. While VM 2 is running, node 2 is active and node 1 can be put into a lower energy mode. As a result, every time we switch between the two VMs, we have to also change the energy mode of the associated memory nodes. While the absolute energy cost for this operation is quite small, it does accumulate over time. If we can instead arrive at a placement as shown in Figure 3.4(b) where we combine the active pages of both VMs in a single memory node, we can avoid the switching and thus save additional energy.



(a) Static Placement Policy (b) Dynamic Placement Policy

Figure 3.4: Different Placements for two VMs

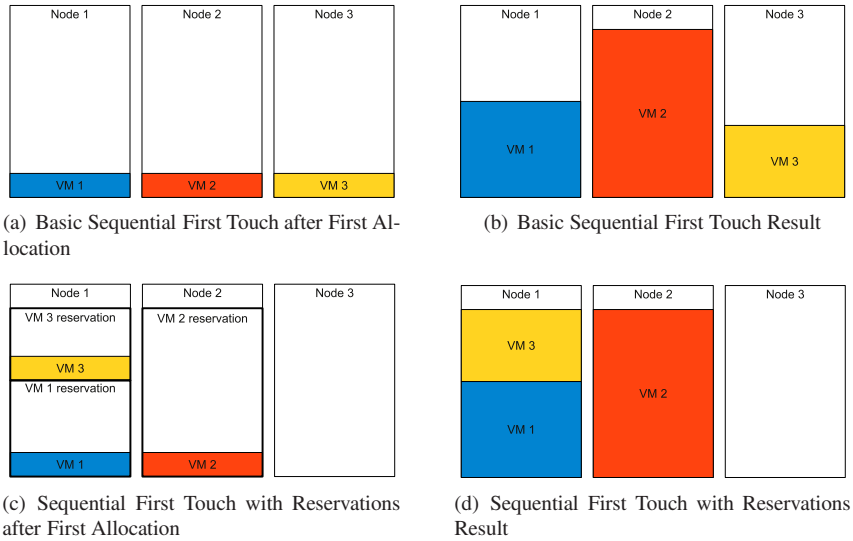
From the example figure we can already observe that transforming the static placement into the dynamic one is an expensive operation. We have to move the whole working set of one VM to another memory node in order to materialize the benefit. In addition, we will most likely have to make room in that node first. Furthermore, if we want to leverage the possible savings from this optimization, we also have to modify the scheduling to consider shared nodes as a criterion for its decision. Otherwise, in a scenario with four VMs in which VM 1 and 2 are sharing a node and VM 3 and 4 are sharing another node, a schedule that runs VM 1, VM 3, VM 2, VM 4 does not save energy at all.

With all that in mind, we conclude that inter-VM migration is not a worthwhile target for our experimentation. The break-even time to migrate a typical working set of some 100MB while also making room on the target node is a matter of hours not minutes or seconds. It is of note that in this case we are looking at a far smaller return-of-investment than in the case above. In the intra-VM case the energy saving comes from powering down a memory node, while in the inter-VM case the saving comes from avoiding to switch power states. The absolute value of this saving is already small and it is only comes into effect whenever the scheduling actually switches between VMs (in common hypervisors between 10 to 100 times per second).

3.4.2 Sequential First Touch with Reservation

Still there is an opportunity for improvement to the sequential first touch allocation policy as described above. In contrast to previous work that is concerned with “normal” processes that allocate and free memory unpredictably, in our scenario we know the

size of the virtual main memory for each VM at start-up (or short thereafter). We can use that knowledge to improve on the initial allocation policy in a way that will co-locate VMs if their virtual main memory is smaller than the node size. As we argued herein above already, with current hardware that provides rather big memory nodes it is likely that we can fit the virtual main memory of more than one VM into a single node.



(a) Basic Sequential First Touch after First Allocation

(b) Basic Sequential First Touch Result

(c) Sequential First Touch with Reservations after First Allocation

(d) Sequential First Touch with Reservations Result

Figure 3.5: Allocation Policy Improvements

The modification to the allocation policy is simple: Instead of choosing the node with the most free memory for the first allocation we choose the node with the least free memory that is still able to hold the whole virtual main memory of the VM we are allocating for. In addition we place a reservation on the node that indicates how much of the nodes memory will be used in the future. Our idea is illustrated in Figures 3.5(a) to 3.5(d). This best-fit policy ensures the most amount of sharing between VMs without causing unnecessary fragmentation.

3.4.3 Idle VMs

In a hardware consolidation setup, we expect that there will be times where many of the active guests are idle for prolonged periods—several hours in some cases. In this section we describe a policy to optimize the handling of idle guests.

While a VM is idle it has a very restricted working set. In the best case scenario the active memory is a single code page that holds the idle loop and a few extra page table pages to map that code page. In this situation it can be beneficial to migrate the working set to the shared system nodes. Considering that the idle loop runs very quickly before it issues a `HLT` instruction that switches back to the hypervisor, it is beneficial to avoid switching memory power states. Alternatively, we can also use a layout as shown in Figure 3.6. The policy copies the few pages that constitute the idle working set to the shared node instead of migrating them. This way we can avoid to migrate back and forth. Instead we have to synchronize the copies if we detect a write access. We can achieve that by mapping these pages read-only in the virtualization

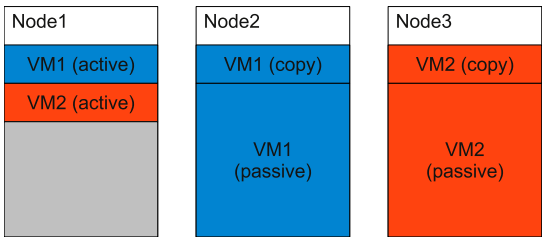


Figure 3.6: Duplication for Idle Guests

page table and handling the synchronization on a write-fault. We argue, however, that write access to pages that are truly part of the idle working set are uncommon.

This optimization pays off almost immediately, because the investment for copying the idle working set is very small. The only challenges are to properly detect that a VM is idle and to find the related working set. Our working set detection already includes all the necessary tools. Once a guest becomes idle, the page reclaim mechanism will quickly shrink the working set. Once the working set size drops below a threshold and the scheduling detects that the guest does not fully exercise the allotted timeslice, the policy can assume that the machine is idle.

3.4.4 Shared Pages

There is a great number of recent work [6, 15, 20] that is concerned with finding pages that can be shared between two or more processes or VMs in order to save physical memory. In a hardware consolidation environment there is a potential for this kind of sharing, especially if several instances of the same operating system and/or application are running on one host. We argue, however, that page sharing between VMs is not beneficial for saving energy in the memory system.

The reasoning behind this is illustrated in Figure 3.7. In this scenario, only VM 1 and VM 2 truly benefit from the placement of shared pages, while VM 3 and VM 4 now need to activate an additional node in order to access the shared pages. We argued earlier, that it is unlikely that more than two or three VMs are located on a single node. In most cases we only have one VM per memory node and thus no direct benefit from sharing with other VMs can be obtained. In addition, page sharing produces additional costs every time a shared page is written to, as we then have to make a private copy of the page.

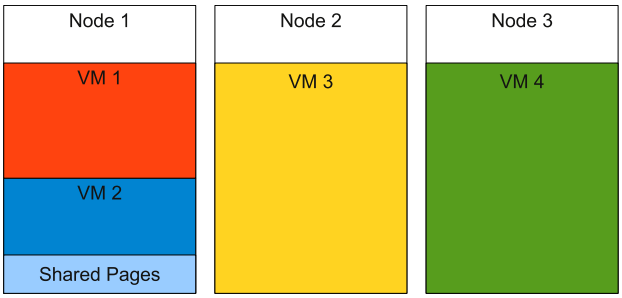


Figure 3.7: Shared Pages

Instead of placing the shared pages in the active node sets of the VMs we could place them in the shared system node. While this allows all nodes to obtain the benefit from the sharing, it produces additional costs. In order to set up a shared page, the responsible process scans all mapped pages. If it finds a page with the same content mapped into two or more different address spaces, it removes all but one copy of the content and remaps all address spaces to point to the remaining copy instead.

If we want the shared page on the system node, we would have to copy the content first. In addition, the system node is a restricted resource, as well. While a single node of 512MB can easily hold the code and data, which is required by the hypervisor, storing shared data in the system node would quickly exceed the size. As a result, we would have to use an additional system node that has to be active for all guests.

It should be noted that para-virtualized approaches to page sharing—such as proposed by Milos et al [15]—can help to save energy in the storage subsystem. Their idea is to track data that is taken off secondary storage and copied to memory, to share the resulting pages directly. This approach can substantially reduce the access frequency to the storage and thus allow the system to switch off the harddisk. This, however, is out of the scope of this work.

3.5 Additional Considerations

In this section we provide additional consideration for two scenarios that are out of the primary scope of our design. First, we discuss how the use of paravirtualization techniques could help to improve our policies. Then we discuss multi-processor and non-uniform memory access systems as the host.

3.5.1 Using Paravirtualization Techniques

There are a number of paravirtualization techniques discussed in related work [20] that can be used to optimize our work further. By paravirtualization we mean techniques that modify parts of the guest operating system in a way that it is possible to communicate information between host and guest system. These techniques aim to either improve policy decisions made in the host by obtaining knowledge only available to the guest, or the paravirtualization helps to improve performance by circumventing expensive emulation.

In particular the concept of memory ballooning can help to improve migration decisions. The idea is to introduce a driver into the guest operating system that can put memory pressure on the guest. It does so by allocating a chunk of memory from the guest. This balloon driver then informs the hypervisor about the location of this memory. In turn the hypervisor now can use this memory otherwise.

For our context, we can use this mechanism to obtain information from the guest operating system. We go through great length to determine which part of the guest's memory is idle. By inflating the balloon, the guest can decide on its own. We could even improve this further. Traditionally, memory ballooning is used to free up memory that is then given to other tasks. Our interest, on the other hand, is to find out the guest's idea of idle memory. If we can construct a paravirtualized driver that can provide us with that information, the guest does not have to give up that memory.

In either case, we end up with a collection of pages that are considered idle by the guest OS and we can use them to make room in an active–active node. In addition we

can use this information before doing a migration in order to avoid moving pages that the guest considers idle already.

Other paravirtualization techniques [2] are mostly concerned with access to devices. By sharing buffer space between host, guest, and even the hardware in some cases, these techniques can improve performance by a great deal. The shared memory that is used in these techniques would need special handling, too. If the hardware is involved, the policy is restricted to certain parts of the physical address space so that DMA can be used and the placement options are limited. Otherwise, future work should investigate where to place these memory buffers to optimize both performance and energy savings.

3.5.2 Multi-Processor Hardware and NUMA

In our design we do not consider multi-processor or non-uniform memory access hardware. We assume that there is only one VM running at any given time and thus only the memory nodes for that VM need to be active. With MP hardware this is no longer the case. Instead we have to schedule the active VMs carefully on the available processors in order to maintain energy savings. In addition we have to consider that scheduling many VMs from the same memory node at the same time on different processors will also mean that the CPUs will congest the memory node. This is bad from a performance perspective. There is previous work [14,17] that can offer some guidance. These works are concerned with identifying congested resources and affecting the schedule in a way to mitigate the resulting performance hit and resource overuse. Future work should consider a VMs active node set as an additional resource to input into the scheduling mechanisms presented therein.

Non-uniform Memory Access hardware does not directly affect our work. For these platforms we can simply divide the system into parts that then can be considered as uniform. As we will show in the following chapter, our implementation borrows mechanisms that are developed to deal with NUMA hardware. We also show that we can extend these mechanisms in a way that leaves the NUMA topology information intact. The only concern for future work in relation to NUMA systems is to find a suitable allocation policy that puts the VMs onto the available NUMA nodes and migrates them between nodes when asymmetric load is detected.

3.6 Summary

In this chapter we have outlined our design for a power-aware memory management for virtual machines. We have modified the sequential first touch allocation policy to our environment. We have also shown our design of a light-weight working set detection and have described a migration policy that optimizes the initial placement done by the allocation policy, using the working set detection as input.

In addition, we have investigated supplemental techniques to increase energy savings. In the course of this investigation, we have excluded page sharing as a viable approach to save energy in our environment. We have also described the impact of our design on systems out of the primary focus of this thesis in order to put our approach into context.

In the next chapter, we provide a prototypical implementation of our basic allocation policy, the working set detection and our migration policy.

Chapter 4

Implementation

In this chapter we describe our implementation of the mechanisms and policies described in the design chapter before. We first introduce our environment and map the abstract terms used during the design stage to the actual entities used in that environment. In Section 4.2 we describe two mechanisms that are prerequisites for the the core of our implementation: selecting the physical location for an allocation, and page migration between memory devices. We then describe our implementation of the sequential first touch allocation policy, the working set detection, and the page migration policy.

4.1 Environment

Out test implementation is based on the QEMU-KVM virtualization suite. KVM [23] is a hypervisor that is hosted inside the Linux kernel [24]. As such, KVM is neither a *type 1* hypervisor that is solely in control of and responsible for the hardware, nor a *type 2* hypervisor that is completely hosted on an operating system as a normal application. We use a recent KVM development tree based off of Linux 2.6.32. This virtualization system is divided into two parts. The QEMU userland emulator package [25] provides full machine emulation, but can work in combination with the Kernel-based Virtual Machine (KVM) hypervisor. This hypervisor lives inside the Linux kernel space. Interaction between the userland and kernel part happens through a pseudo device and a number of control operations (`ioctls`) that can be issued on the device.

4.1.1 Starting a VM

In order to start a VM, we start a QEMU userland process. This process first asks KVM to create a VM inside the kernel. At this point KVM allocates the main resources to hold the VM's state and bookkeeping. The QEMU process then maps virtual memory to back the VM's virtual main memory using the `mmap()` system call. At this time no memory is allocated to back this virtual memory. The QEMU process then informs KVM about size and (virtual) location of the virtual main memory using the `/dev/kvm` pseudo device. It also informs the hypervisor about the guest physical location that this mapping refers to. The hypervisor adds the information to the VM's resources for reference later on. The QEMU process also maps some of its code as virtual main memory. This code is responsible for basic emulation such as BIOS func-

tions. Finally—after some more basic set up operations—the QEMU process asks KVM to start execution of the VM. KVM sets up the CPU for virtualization and starts execution of the boot code.

As the boot sequence processes and touches portions of the virtual main memory, the KVM page fault handler is asked to resolve mappings of previously unmapped guest physical frames. The page fault handler uses the processes described in the background chapter. First the hypervisor needs the host virtual address inside the address space of the QEMU process that backs the requested guest physical page frame. KVM locates this information with the data previously obtained through the pseudo device. Once the virtual address is known, KVM resolves it to a host physical address using the page table that manages the address space of the QEMU process. If this is the first time that this page is touched, the kernel allocates a physical page frame at this point. Otherwise, the PTE in the page table of the QEMU process contains the host physical address. Now that both guest physical and host physical address are known, KVM can enter the mapping into the virtualization page table and the virtualization can continue. Subsequent accesses to the same guest physical frame will no longer cause a page fault.

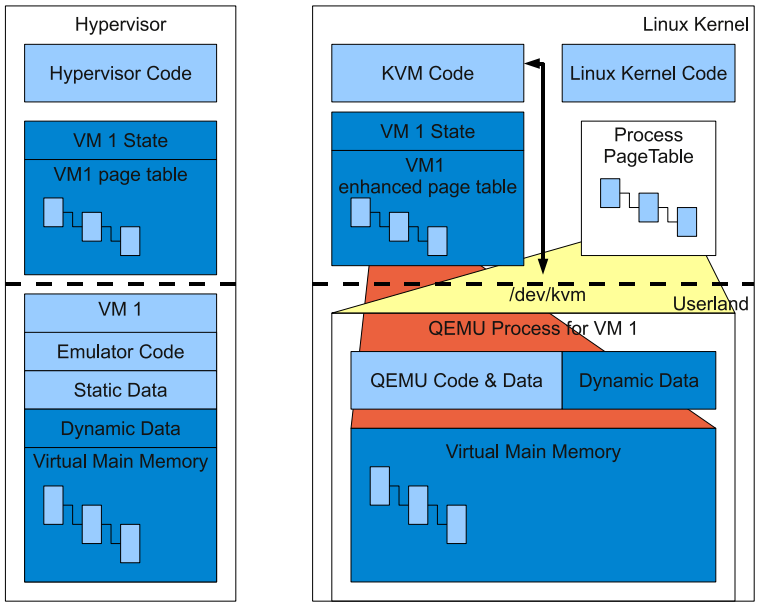


Figure 4.1: Comparison between the abstract concepts used during the design chapter and the actual entities in the implementation environment

Figure 4.1 illustrates how the concepts from the design chapter are mapped to actual implementation details. In this picture we revisit the abstract concept of different kinds of memory and compare them to actual entities in our implementation environment.

4.2 Prerequisites

In order to implement our placement strategy we have to control on which node an allocation request ends up. In addition, we need a mechanism to move a mapped page from one physical location to another.

4.2.1 Leveraging the Linux NUMA Framework

In order to support *Non-Unified Memory Access (NUMA)* systems, the Linux kernel already includes a framework that provides support for the required mechanisms. In a NUMA system there are also memory nodes to consider. In contrast to our work, these nodes are not power-management units, but memory that is attached to a (set of) CPU(s). The goal of a NUMA aware allocation policy is to allocate memory for a task—running on a certain CPU—from a memory node that is close to that CPU(s) in order to avoid delay.

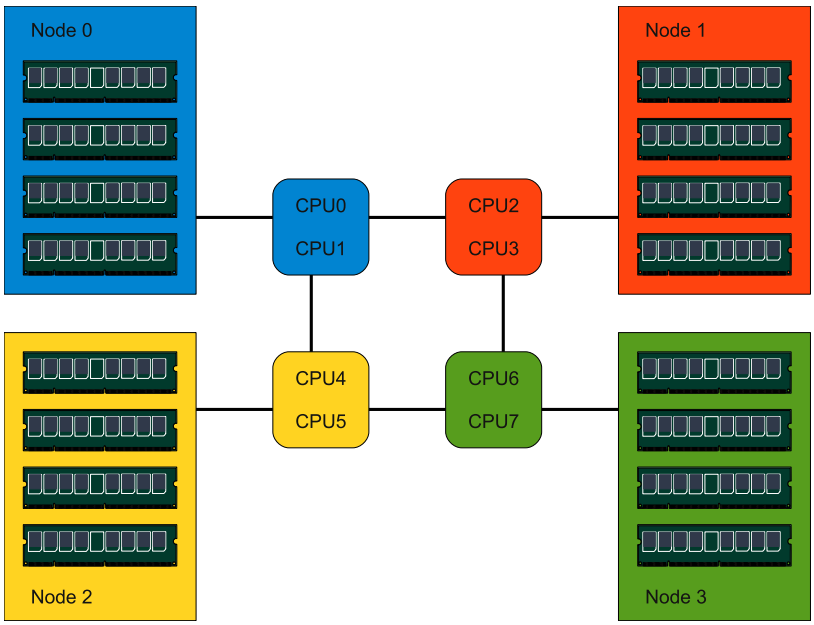


Figure 4.2: Example NUMA Layout

Figure 4.2 shows an exemplary NUMA topology with four CPU packages—two cores each—and attached memory nodes. Due to the simplistic interconnect between the individual packages, there are different NUMA distances to the memory nodes of different CPUs. For example, CPU0 has a distance of zero to its own node 0, a distance of one to node 1 and node 2, and a distance of two to node 3. For an allocation on behalf of a task that runs preferably on CPU0/1 the NUMA policy would prefer memory from node 0 over memory from node 1/2 over memory from node 3.

For our work we extend this framework by splitting a single memory node (in the NUMA sense) into its power-manageable units. Figure 4.3 shows the result. We keep the notion of NUMA distance, however. Doing so will allow future work to consider NUMA and multi-processor systems. As a result, we end up with a set of (NUMA) memory nodes that represent power-management units. Using this setup, we can apply specially crafted NUMA allocation policies that consider power-management units.

The NUMA framework allows the system to attach an allocation policy to a number of kernel entities. We can define a policy for a whole process, but we can also attach a policy to a portion of a process' address space. We will use the latter to attach our Sequential First Touch policy to the parts of the QEMU process that map the virtual main memory. In addition, the NUMA framework allows us to define a default policy.

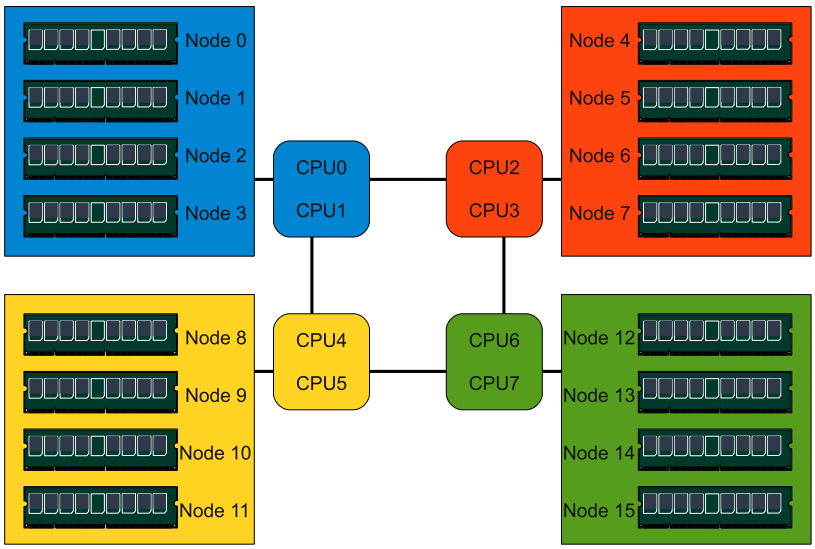


Figure 4.3: Power-Aware NUMA Layout

We use this mechanism to put all allocations, which are not for virtual main memory, onto the first node(s)—the shared system node(s).

The NUMA framework also includes a primitive for page migration. A single function—`migrate_pages()`—takes care of all the details. The input for this function is a list of pages to migrate and an allocation policy to find a new location for the migration. The function takes care of all the details required for the migration. To migrate a single page, it first unmaps the page from every address space that has a mapping to that page. It then calls the provided allocation policy to find a new destination for the contents of the page. Finally, it copies the page to the new location and remaps the page into the affected address spaces.

As NUMA hardware is on the rise, we expect that any modern hypervisor will also have to provide some type of handling for these systems that can be used in a similar fashion to implement our changes on other systems.

4.3 Implementation

With this background we now describe the actual changes to KVM to implement the policies described in the design chapter. Due to time constrains we did not implement all of the ideas presented therein.

4.3.1 Sequential-First-Touch Allocation Policy

The first building block, which we implement, is the basic *sequential-first-touch* allocation policy. With the help of the NUMA framework this was quite simple and straight forward. As we mentioned earlier, the NUMA framework allows the kernel to attach a policy with a process or a virtual memory area (VMA) inside the address space of a process. Whenever a physical allocation for that task or VMA is required we end up in a central function inside the NUMA framework. At this point we have access to a data

structure that describes our policy. We can then use primitives that allocate a page on a selected node to satisfy the requested allocation.

In the data structure that describes a sequential-first-touch policy, we only have to store the active node set. We define a helper function `pick_node()` that takes the active node set as input and returns the node from which we should allocate next. This function performs the policy as outlined in the design. Checking for the node with the most free memory is an expensive operation. It requires synchronization and—the way it is implemented in the Linux kernel at this time—a memory allocation and copy. That is why, we modify the policy slightly for better implementability. Instead of choosing a new node at every allocation, we store the last node that we used in the policy data structure. `pick_node` returns this node for subsequent allocations without having to check for free memory again. Once there is no more free memory in the node, the subsequent allocation primitive fails. At this point the allocation function calls `pick_node` again, this time forcing a check for free memory. This change does not impact on the outcome of the policy, but significantly speeds up the process.

With this, we only need to attach the same policy data structure with all VMAs that describe the virtual main memory of a single VM. This is easily done in the `ioctl` that QEMU uses to inform KVM about the location and size of the virtual main memory. We simply transform the information provided by QEMU into VMAs and attach the VM's policy data.

4.3.2 Working Set Detection

The basic implementation idea for the working set detection is already described in Chapter 3. We consider all page frames that KVM has mapped in the virtualization page table to be part of the working set. This makes it easy to obtain the information that we need. All we have to do, is to add an accounting mechanism into the page fault handler. Every time a mapping is added to the virtualization page table, the accounting looks up the node, to which the host physical address belongs to, and increases the active page count for that node. Without further changes this would result in a working set that contains all pages that were ever touched. By default, KVM allows the virtualization page table to grow until all guest physical addresses are mapped.

In order to obtain a current working set instead, we limit the size of the virtualization page table. As a result KVM now has to reclaim old mappings before it can add new ones. We describe how KVM does the reclaim in the following section. By using a limited VPT and reclaiming old mappings, the mappings in the VPT represent a true picture of the working set. It is this reclaim mechanism that introduces the performance concerns that we discussed during the design. If a VM has a current working set that is bigger than what can be covered by the restricted virtualization page table size, the reclaiming will result in a lot of page faults. In order to control the performance loss, we have to limit the rate of page faults.

Virtualization Page Table Page Reclaim

The KVM code already includes means to reclaim pages allocated to the virtualization page table in order to react to memory pressure. The existing implementation is meant to be a least-recently-used approximation. Unfortunately—probably due to the fact that the memory pressure hook is little exercised in normal operation—there were several bugs in that mechanism. We replaced the existing code with a true LRU approximation using a clock algorithm.

The basic idea is to add each virtualization page table page to the head of a list of all VPT pages upon allocation. When we want to find a candidate for reclaim we scan that list from the tail end. For every page we check the referenced bit on that page's parent PTE. If a reference has occurred since the last scan, we clear the reference bit and move the page to the head of the list. Otherwise we have found a candidate. In order to avoid overscan, we limit the scan window to find a single candidate to a quarter of the whole list. If no candidate has been found in that window, we reclaim the first page that is located at page table level. The reasoning behind this being the fact that reclaiming a page at page directory level page would tear down a whole section of the guest physical mappings that might still be active.

During this work we discovered that Intel's implementation of the virtualization page table does not set the referenced bit in the EPT page table entries. This defeats all efforts to implement LRU. In order to overcome this problem we use the PTE's access bits instead. Now we check if the parent PTE has the access bits set and clear them instead of the reference bit before moving the page. This means that we now have to take an additional page fault due to the access violation on the next reference to that page. Fortunately, this kind of page fault can be handled quite quickly as we only have to walk the EPT in software up to the point where the missing access bits are and set them. This is far from optimal, but does not impact our experimentation too much. In contrast to Intel's EPT, AMD's nested page tables do set the referenced bit. It is our hope that Intel will follow suite in future CPU revisions as this feature is helpful for other tasks as well.

Limiting the Page Fault Rate

With the reclaim mechanism in place we can now leverage it to complete the working set detection. Again, our goal is to balance the working set size against the rate of page faults that are caused due to the analysis. We start off with a limit of 64 VPT pages. This is the suggested minimum as otherwise the mapping might deadlock while doing the nested page walk. 64 VPT pages are enough to map up to 122 MB of linear guest memory, which is already a relatively large working set to consider. Due to safety margins and the fact that we rarely access the guest virtual memory in a linear fashion, the actual perceived working set size with 64 VPT pages is usually between 30 and 60 MB. This reduced size is small enough to handle for migration.

Now we need to balance this against the page fault rate. Instead of actually measuring the page fault rate that would also include the page faults due to the page reclaim hack described above, we instead gauge the page reclaim rate. This is indicative of the actual page fault rate and happens less frequent. If this rate exceeds a threshold[1] we do not reclaim a page, instead we increase the number of VPT pages that we allow to be allocated at a time. We limit the growth of the VPT to 50% the size that would be required to map the whole guest virtual main memory, as we consider a working set of more than half the available memory to be in violation of normal guest behavior.

With these changes that arise from our implementation of the working set detection, we were able to successfully counter act performance degradation due to the working set detection. However, if we only ever grow the working set, we will later have a problem when we try to find a manageable part of the working set for migration. In order to avoid this, we also need a mechanism to shrink the working set once the page fault rate is back to a reasonable amount. Once the page fault/reclaim rate stayed below

[1] We find a page reclaim rate of less than 128 per second to be a good threshold during the evaluation.

the threshold for some time[2] we use the page reclaim mechanism to free up all VPT pages that have not been accessed during the last two seconds. If the page fault rate stays below the threshold, we can assert that the working set has indeed become smaller and we can reduce the number of pages allocated to the VPT again.

4.3.3 Migration

For our experimentation, we only implement intra-VM migration. As discussed in the design chapter, taking advantage of inter-VM migration requires substantial changes to the scheduling that could not be implemented during our timeframe.

We decided that the migration should take place inside a background task instead of inside the processing path of the virtualization. This approach avoids a too big impact on the performance of the VMs. We created a kernel level thread that is in charge of the migration. This thread is given a list of all active VMs. It cycles that list and performs the migration policy for each of them before going to sleep for a while[3] and starting over.

We implement the migration policy in two stages: The first stage checks the current VM's working set for old nodes as defined in the design. The second stage of the migration process takes care of the actual migration.

If no old node is found, the migration thread also takes care of shrinking the working set. It first checks if the VM occupies more than a single node—otherwise shrinking the working set does not provide any benefit. If more than one node is active–active for the VM, it then checks the recent page reclaim rate for the VM and—if the rate is below the threshold—it reclaims a portion of the VPT pages. After doing the reclaim, the process continues with the next VM in the list.

If the scan identifies an old node, the process continues by finding a target node for the migration. Once we have a possible source and target node, we then check if there is enough room in the target node to perform the migration. We then calculate the break-even time for the migration based on the findings. If there is enough room, we only need to consider the number of active pages in the source node. Otherwise we have to add the number of pages that we have to move out of the target node to the calculation. Once the break-even time is determined we compare it against the source node's age. If the node age is greater than the break-even time, we then move to the next stage with source and target node, as well as the number of pages that we need to move away from the target node, as input.

The first task of the second stage is to make room in the target node, if the previous stage so requested. In order to do that, we rely on the default Linux LRU mechanism that is used to find candidate pages to swap out to secondary storage. We ask this mechanism to provide us with the desired number of least-recently-used pages in the target node. This mechanism suffers from the same problem as our VPT LRU implementation: The EPT page lookup does not set the referenced bits and thus the LRU information is flawed. In order to combat this problem we use a hack that was proposed on the Linux Kernel Mailinglist:

Before the Linux LRU scanner adds a page to its candidate list for swap-out, it issues a mmu-callback to interested parties. This is already used by KVM in order to unmap VPT mappings for the page. To avoid that a page that is mapped in the VPT

[2] We wait for one second without a page reclaim event and fewer than 64 page reclaims in the previous second for this threshold.

[3] We find that a five second interval provides good results for the experimentation.

hierarchy is added to the candidate list, we use that callback to indicate that the page is still in use and the scanner moves on to find other candidates.

Once we have a list of least-recently-used pages, we then migrate these pages away from the target node. We use the normal allocation policy to find a new home for these pages—explicitly avoiding a reuse of the target node.

If the migration succeeded, or we had enough room in the target node to begin with, we then have to identify the active pages on the source node. To do so, we walk the VPT in software comparing the mapped host physical addresses against the source node. Each matching mapping is removed and the backing page is added to a list. Once all pages are collected, we migrate the pages in that list to the target node and the migration is complete.

Old Node Detection

As we mentioned in Section 3.3.1 of the design, it is infeasible to track the age of every mapping in the VPT due to the performance and space impact. Instead we consider the most recent mapping in each node as the age of that node. To reduce the impact of this process even further, we move the age calculation of a node to the migration task. This avoids taking a timestamp, which itself is an expensive operation compared to the optimized page fault handling process. In order to do that we store the number of active pages in each node on every migration scan. If the number of active pages for a node has not increased since the last scan, we increase the node age by the migration scan interval. While this does not give an exact value for the node age, it provides a good lower bound to use for the migration decision.

We base the migration interval on half the break-even time for a usual migration size. Preliminary experimentation has shown that we usually migrate about 10 to 40 MB worth of data independent of the workload. This corresponds to ten seconds for the break-even time—respectively five seconds for the migration interval. While this seems rather intrusive on first sight, remember that most of the time we only check the working set data, conclude that a migration is not required, and return. In case we do not actually make a migration decision, the processing consists only of a few loads and branches.

Chapter 5

Evaluation

In this chapter we use our implementation, which we described in Chapter 4, in order to evaluate our design. We show two things in the following: First we show that our changes to the memory management do not impact on performance in a negative way. In fact, we show that there is no measurable difference. Then, we provide an approximation on how much energy can be saved with our changes. As we discussed already, we were unable to directly measure energy savings due to hardware shortcomings. Instead we use a trace driven analysis to provide a lower bound for the possible savings.

In the following sections, we first describe the hardware platform for our experimentation. We then explain how we obtain trace information for analysis further down. We also present a simple memory energy model based on the data that we obtain from the traces. In Section 5.3, we describe the different workloads that we use during the analysis. We present performance measurements for a number of benchmarks, comparing the unmodified baseline with our changes, in Section 5.4. Finally, in Section 5.5, we evaluate possible power savings based on the energy model.

5.1 Hardware Setup

We use an Intel Core i7 system. It was our initial hope that this setup will not only provide us with the required advanced virtualization features, such as the enhanced page table, but also allow for hardware memory power-management. The memory controller embedded into the Core i7 CPU provides three independent channels with up to four memory nodes each. Unfortunately, the controller makes use of memory interleaving by default. As a result, a single physical page frame is distributed over all available memory nodes and we can not switch off any nodes. We tried to disable the memory interleaving with the help of MSI—the vendor of the mainboard for our evaluation system. Unfortunately, these efforts were to no avail, despite much help from the vendor for which we are truly thankful.

Our system is equipped with three memory devices of 2048MB. We use the NUMA framework to divide the resulting 6GB of main memory into nodes of different sizes. This way we can evaluate our changes for different memory layouts.

5.2 Trace Driven Analysis

Our goal is to compare the energy consumption of the system with our changes to that of the unmodified base. We want to simulate a simple, reactive hardware power management based on the access patterns that result of the default allocation policy.

In lieu of measuring the actual power consumption of the memory system, we would like a trace of the memory accesses that the host system performs on behalf of the guests. With that information we can then drive a model to arrive at the required power for the observed access pattern. Unfortunately, it is not easy to obtain this information without incurring a massive performance penalty that results in a completely different run-time behavior. Otherwise we would use the memory access trace to drive our working set detection, as well. Instead, we have to find an approximation. Our working set detection is one possible approximation that we can use. However, we must make sure that we avoid overestimation of the working set as this would unfairly bias the analysis towards our design. If we overestimate the working set we would consider more nodes active than are actually in use. Thus we explicitly choose the trace points and analysis in a way that we underestimate the working set. If we can show energy savings with an underestimated working set we can be sure that the actual savings will be greater.

5.2.1 Trace Implementation

To obtain trace events from the hypervisor in the kernel, we need a facility that exports the trace data to the userland. Fortunately, the Linux kernel already includes a powerful tracing framework. This framework allows us to define trace records that are stored in a circular buffer inside the kernel. This buffer is exported to userland where it can be read and written to secondary storage for analysis. The mechanisms used to generate the trace records and to export the data to userland are designed extremely lightweight. In addition we use the multi processor capabilities of our experimentation system to avoid most of the impact of the tracing on the experiments. We do so by pinning the VMs and all related tasks and processes on a single processor (as we want to simulate a single CPU for the experiment). The export of the traces takes place on a different processor that is mostly independent of the one where we run the experiment.

5.2.2 Trace Events

For our model, which we will describe in more detail further below herein, we need to trace the following events:

VM Scheduling We need to know when and which VM is scheduled to run on the CPU.

Memory Accesses We need to know the physical address (or at least the memory node) for each memory access that ends up on the memory bus.

Migration We need information on the migrations we perform.

The first type of trace—the scheduling information—is already implemented in Linux and provides all the necessary data: A timestamp, and the process ID for the previous and the next running task.

Exporting information about the activities of the migration task is also a simple matter of defining a new trace record that holds the required information and calling

a special function in the migration task. The information that we need to drive the model consists of: The VM for which we are migrating, the number of pages that are migrated, and the source and target node.

Tracing memory accesses—on the other hand—is a more complex problem. We considered a number of possible approaches to obtain this data.

Performance Monitor Counters Recent processors provide a number of special registers [22] that can be programmed to count performance critical events in order to profile a workload. In particular, there is a performance event called "Last Level Cache Miss" that provides exactly the information we are interested in. This event happens whenever a memory access can not be satisfied by the cache hierarchy and thus ends up on the memory bus. Unfortunately, counting this event alone does not provide us with enough information for our trace. We need the physical address for each event of this kind. This information is not provided by the performance counter. We can, however, define a threshold for the counter. Once this threshold is reached the hardware issues an interrupt and we can look at the processor state to figure out the instruction that caused the memory access. Again, this information—while valueable for performance profiling—does not help our analysis as the cause of the actual memory access can be any number of things. Either it is the instruction itself that is not cached, or it can be any of the operands of the instruction, or it can be the page table information to back any of these. In summary, while Performance Monitor Counters are a valueable tool for some situations, the current implementation of them does not help our analysis.

Page Fault Driven Analysis As with our working set detection, another possible way to obtain memory access patterns is tracing page faults. This has the advantage that—in order to handle the page fault—we already know the physical address for the access. The backdraw is that a page fault does not necessarily correspond to a real memory access. Especially if we try to trace all memory accesses by eagerly unmapping data, the data might still be stored in the cache hierarchy and the access does not end up on the memory bus. In addition, handling page faults is an expensive operation. We specifically designed our working set detection to avoid unnecessary page faults for that reason.

For the analysis we use a page fault driven approach. In fact, we use accounting that is very similar to our working set detection. In order to avoid overestimation of the working set, we employ a more aggressive page reclaim strategy. While the page reclaim rate due to page faults is below 128 per second, a thread—running every 500ms—reclaims all VPT pages that have not been referenced during the previous 500ms. In addition, we consider the VPT pages themself to be part of the working set. This is different from the normal working set detection, where we keep the VPT pages on the—always active—system node. This approach seems to bias the analysis towards our modification, at first. Remember, however, that we do account for the system node when we calculate the energy consumption of the system with our changes. In contrast to that, in the baseline accesses to the hypervisor code and data are not recorded or accounted for.

We define a new trace record to track the working set as perceived in the hypervisor. The event record contains: A timestamp, a node identification and the information whether the node is added to or removed from the working set.

5.2.3 Energy Model

We use a similar model as used in previous work [8]. The basic idea is to use the memory access patterns obtained through the trace and infer the power state of the individual nodes from that information.

U	Set of all guests in the system
V	Set of all memory nodes in the system
$t_{i,j}^{active}$	Total time that node j is active while guest i is running
$t_{i,j}^{idle}$	Total time that node j is idle while guest i is running
m_i	Total number of migrations performed on behalf of guest i
P_{active}	Power dissipation of a node in active power state
P_{idle}	Power dissipation of a node in active power state
$E_{migration}$	Energy required to migrate a single page between two nodes

Table 5.1: Input Data Used in the Energy Model

The static energy used in the memory system can be calculated by a simple formula:

$$E = \sum_{i \in U} \sum_{j \in V} (t_{i,j}^{active} P_{active} + t_{i,j}^{idle} P_{idle}) + \sum_{i \in U} (m_i E_{migration})$$

The data used in this model is shown in Table 5.1. V, P_{active}, P_{idle}, and $E_{migration}$ are parameters of the system setup and provided to the analysis as such. The analysis gathers information about the running guests $\in U$ through the scheduling traces. We calculate $t_{i,j}^{active/idle}$ and m_i based on the trace data. For every guest $\in U$ we keep the set of active nodes used by the working set of the guest. For every scheduling event in the trace that starts or stops a guest, we modify the active and idle times for that guest based on the current active node set. For every working set change event, we recalculate the idle and active times based on the current active node set, before we modify the active node set for the affected guest. The final parameter (m_i) is directly reported in the trace event.

In the referenced article [8], the authors calculate the total energy instead of just the static energy. They do so by calculating a *memory activity factor* obtained through performance counters. The activity factor is calculated by dividing the number of observered memory transaction by the number of possible memory transactions in a given time interval. This factor is then multiplied with the power dissipation for a read or write operation. We argue, however, that this calculation does not reflect the actual energy usage, as the actual energy requirement for read and write is again dependent on the node that is the destination of the operation. This is not reflected in the model. In addition, our changes have no impact on the dynamic energy consumption and thus we do not consider it for our analysis.

5.3 Benchmarks

In this section we describe the workloads used during this evaluation. We set up a number of guest machines with freely available open source operating systems. Inside the guest machines we run two different types of workloads. We argue that the main difference for memory behavior in our environment is caused by the guest operating system and not by the workload that runs inside the guest.

5.3.1 Workloads

In order to run the same workload on different operating systems, we selected the NetBSD kernel build as the first workload. The NetBSD build systems allows us to build the same compiler suite and toolchain on any target. The build system then uses the previously compiled tools to build a kernel. This way our evaluation can focus on the differences in the guest operating system that runs the build. A source compilation workload is a good approximation of a typical mixed workload: Some of the tasks performed during a build are I/O-bound. For example, dependency tracking and preprocessor runs. The actual compilation is CPU-bound, while linking, symbol resolution and some optimizations are memory-bound.

As the second workload, we use a preconfigured guest image that can run a selection of SPEC CPU2006 [26] benchmarks: 429.mcf, 444.namd, 458.sjeng, 470.lbm, 435.gromacs, 456.hmmer, and 462.libquantum. A detailed analysis of the memory footprint and access behavior of these benchmarks can be found in an article by Gove from 2007 [7]. The memory access patterns resulting of these benchmarks are distinct and help to test our migration policy against a number of different cases: Both fast and slow sequential, as well as random access to large and small working sets.

5.3.2 Guest Operating Systems

For the NetBSD source compilation, we use three different guest operating systems:

Debian 5.0.3 x86_64

FreeBSD 7.2 x86_64

OpenSolaris 2009.06 x86_64

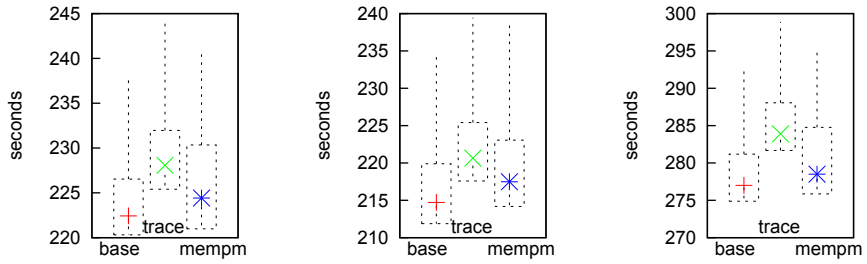
We installed all guest on a 8GB pre-allocated raw disk using the original installation CD and default settings. In addition, each guest is given a 1MB *configuration* disk that we generate on the host. This disk contains the sequence of benchmarks that we want the guest to perform. In addition, we can define a sleep time between individual runs inside the guest. On startup, the guest reads the information from the configuration disk and performs the benchmarks accordingly. Once all work is done, the guest writes the performance information—obtained by the `time(1)` utility—for every run into the configuration disk and shuts down the VM. This setup allows us to perform any sequence of benchmark runs in an automated fashion, and to obtain the performance information on the host through the configuration disk.

In addition we use a previously set up Linux 2.6.26 x86_32 installation that runs the SPEC CPU2006 benchmarks described above. We use a similar setup as for the other guest in order to run any sequence of SPEC benchmarks with additional sleep time between individual runs.

5.4 Performance

In Chapter 4, we identified two configuration parameters for our implementation that can impact on performance: The page reclaim rate and the rate at which we resize the virtualization page table in case we detect a low page reclaim rate. We choose these parameters in order to have minimal impact on performance. We started with a page reclaim rate of 1024 per second and divided by two until we did not see any

measureable impact on our benchmarks. Using this method we arrived at 128 page reclaims per second as the threshold. In addition, we found that resizing the VPT in face of a low page reclaim rate did not impact performance significantly. This is of particular interest, as we attempt to resize the VPT every 500ms for the baseline trace comparison.



(a) Debian (b) FreeBSD (c) OpenSolaris

Figure 5.1: NetBSD Kernel Build Performance under different guest operating systems

Figure 5.1 shows the NetBSD kernel build times in the different guest operating systems. Each plot gives the median[1], upper and lower quartiles[2] centered around the average, as well as minimum and maximum for each environment. These numbers are calculated from 25 runs in each environment. The "base"-environment is the unmodified KVM source, "trace" refers to the environment used for comparison in the following section, and "mempm" denotes our changes. The slight absolute difference between "base" and "mempm" in the figures is due to the tracing, not the actual changes to the memory management. The fact that the difference is only about 1% and in some cases not even of statistical relevance, shows that the trace is truly lightweight and does not impact on the runtime behavior.

The difference between the "mempm" and "trace" environment is due to the more aggressive VPT resizing. The penalty for the traces is 1.3% on average. We account for this penalty, by modifying $t_{i,j}^{active/passive}$ accordingly when we calculate the energy consumption in the "trace" environment.

Figure 5.2 shows the performance of the various SPEC benchmarks. In these plots we use the arithmetic average over all runs as the reported time already excludes all interaction with secondary storage. Again, the difference between "base" and "mempm" is below 1% and in most cases not of statistical relevance. The mcf benchmarks exhibits the biggest variance. This is also the benchmark with the biggest memory footprint, and the most memory bound one. CPU bound benchmarks (e.g. gromacs), in contrast, exhibit very little variance.

The difference between "mempm" and "trace" is between 1.7% and 2.8% and we account for this difference when calculating the energy consumption below.

5.5 Power Savings

We use the trace data for different benchmark runs and the memory model from Section 5.2.3 to plot the cumulative static energy used during a run. The plots are given relative

[1] We prefer the median over the arithmetic average in order to exclude the first run that populates buffers, instead of manually excluding these runs.

[2] We use the t-distribution to calculate quartiles.

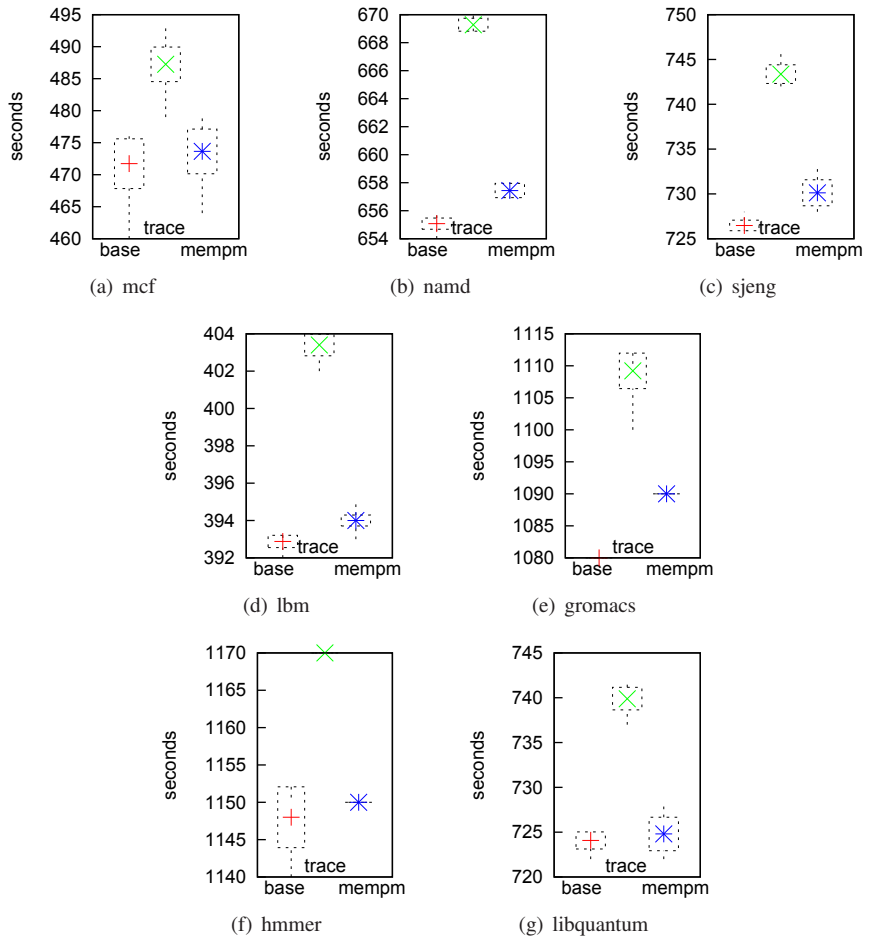


Figure 5.2: Performance of the different SPEC benchmarks in our three test environments

to the maximum static energy when all host nodes are active all the time. We compare to that, the static energy use with the baseline and with our changes. In the following, we discuss different scenarios and point out where and how our changes save energy.

5.5.1 SPEC CPU2006

The plots in this section refer to a benchmark run with a single guest running the SPEC benchmarks. The guest is given 4GB of virtual main memory. After each individual SPEC run, the guest sleeps for 60 seconds before starting the next run. We run the tests in the same order as mentioned above: 429.mcf, 444.namd, 458.sjeng, 470.lbm, 435.gromacs, 456.hmmer, and 462.libquantum.

Figure 5.3 shows the cumulative energy used for the SPEC tests on a host with twelve memory nodes, 512MB each. In this test case, our modification uses over 60% less static energy than the maximum, and about 29% less than the baseline. In addition, this test case shows the effect of the migration. 2600 seconds into the run "mempm base" needs to add another node to house the virtual main memory and the energy

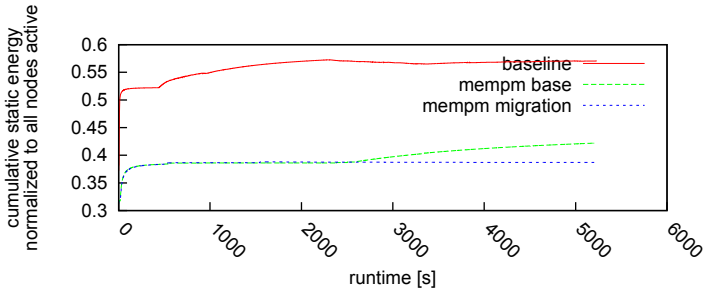


Figure 5.3: A single guest with 4GB of virtual main memory running all SPEC CPU2006 benchmarks, spaced by 60 seconds with 512MB nodes

consumption starts to increase. In contrast to that, "mempm migration" manages to keep the virtual main memory on fewer nodes and saves additional energy.

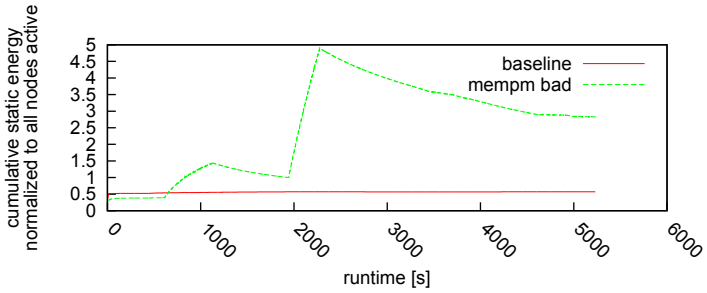


Figure 5.4: Same test run as 5.3 showing an implementation error in an early version

Figure 5.4 show the same test case as above, but with an early version of our implementation. This version had a faulty implementation of the exponential back-off mechanism that we discuss in Chapter 3.3.2. As a result the migration policy continuesly migrates a small amount of memory from one node to another, but the source node of the migration does not stay idle long enough to pay for the migration cost. The energy consumption quickly grows to up to five times the basic—all nodes active—energy usage. We only found this bug when we ran the complete SPEC benchmarks as the gromacs benchmark—starting around 2000 seconds into the run—exhibits a memory access pattern that aligns with the break-even time.

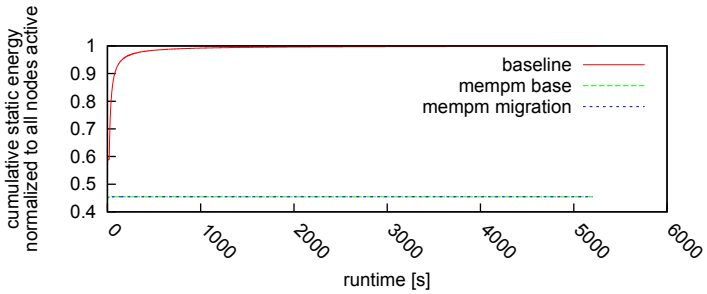


Figure 5.5: A single guest with 4GB of virtual main memory running all SPEC CPU2006 benchmarks, spaced by 60 seconds with 1024MB nodes

In addition, we ran the same test on a host configuration with six nodes, 1024MB in size (cf. Figure 5.5). While in this scenario the modified version still reduces the static energy consumption by 55%, the baseline consumes as much energy as the maximum—all nodes active—case. The reason for this is that the baseline allocation policy now has fewer nodes to choose from. As a result, the—in-effect—random node selection of the buddy system allocator is more likely to place at least one page of the working set into every node. Hence the baseline must keep all nodes active. In this case there is no difference with migration, as no additional node is required late in the run.

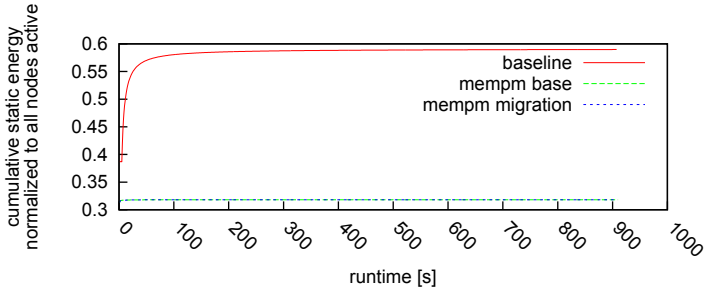
5.5.2 NetBSD Source Compilation

In this section we discuss benchmark runs of the NetBSD kernel compilation with the different guest operating systems.

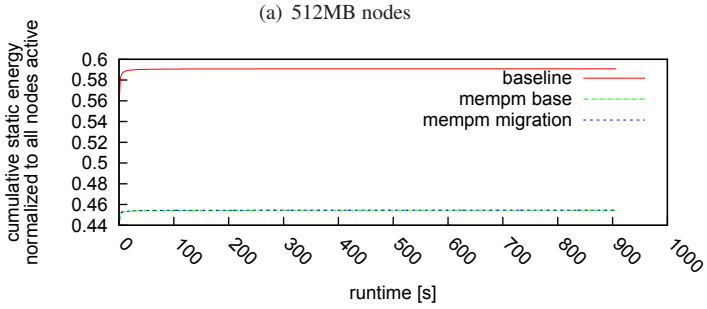
Debian

Figures 5.6 shows the cumulative energy consumption of a single Debian guest running the NetBSD kernel compilation test. The VM is given 4GB of virtual main memory and the host is setup with twelve nodes of 512MB and six nodes of 1024MB.

In the default setup, Debian uses very little memory for the buffer cache. As a result, the test has a memory footprint of under 512MB. That is why "mempm base" and "mempm migration" do not differ in these plots. The guest only uses a single node. The resulting energy savings are 67% for the 512MB nodes and 55% for the 1024MB nodes. The baseline is at about 59% of the energy consumption with all nodes active for both cases.



(a) 512MB nodes

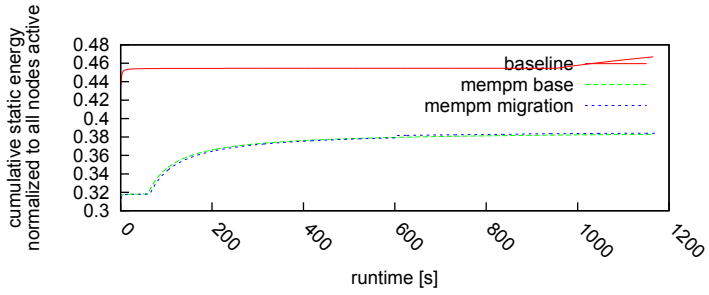


(b) 1024MB nodes

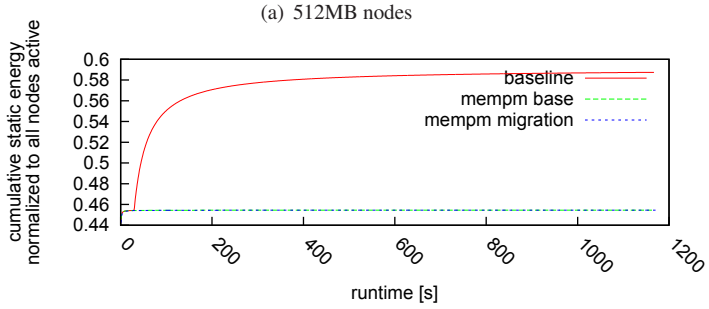
Figure 5.6: A single Debian guest with 4GB of virtual main memory running the NetBSD source compilation spaced by 60, 120 and 240 seconds

OpenSolaris

The OpenSolaris compilation runs, show in Figure 5.7, are similar to that of Debian. The only difference is due to the slow start of the system. In addition, OpenSolaris exhibits a slightly less random access pattern that reduces the energy consumption in the baseline.



(a) 512MB nodes



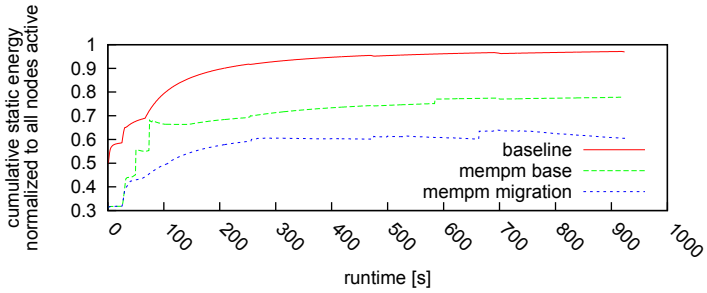
(b) 1024MB nodes

Figure 5.7: A single OpenSolaris guest with 4GB of virtual main memory running the NetBSD source compilation spaced by 60, 120 and 240 seconds

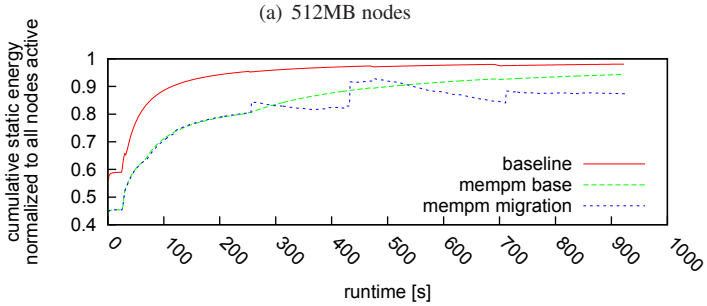
FreeBSD

Figure 5.8 shows the source compilation benchmark, this time with FreeBSD as the guest operating system. In contrast to Debian and OpenSolaris, FreeBSD uses all available memory as buffer cache in the default configuration. Every block that is read from or written to secondary storage is cached in memory. While some of the cached data is reused—and thus remains part of the working set—other parts of the cache are written once and never touched again. In this situation, migration is most effective. The result is up to 20% less energy consumption with migration enabled in the test case with 512MB nodes.

The test case with 1024MB nodes (cf. Figure 5.8(b)) illustrates the break-even time. Initially, both modified versions use comparable amounts of energy until—250 seconds into the run—the first migration is performed. Due to the bigger nodes, the number of pages that need to be migrated is relatively big. We see a notable spike in the energy usage of “mempm migration” caused by the migration. At this point the energy consumption of the version with migration is higher than that of “mempm base”. During the following 100 seconds, however, the energy consumption of “mempm migration” drops as the migration successfully decreased the number of active–active nodes. In contrast to that, the energy consumption of “mempm base” continues to rise. About



(a) 512MB nodes



(b) 1024MB nodes

Figure 5.8: A single FreeBSD guest with 4GB of virtual main memory running the NetBSD source compilation spaced by 60, 120 and 240 seconds

50 seconds after the migration, the energy consumption for both cases are equal—the break-even time has been reached. At second 450, we see another energy spike due to migration. This time even more pages than before are migrated. We also see that this first migration was unsuccessful—the energy consumption for "mempm migration" keeps rising after the spike. This means that additional pages from the source node of the migration have become active and we enter the exponential back-off. Another 20 seconds later, the back-off time has passed and the policy attempts another migration. This time only a few pages are migrated, but the migration is successful and from that point forward the energy consumption for "mempm migration" drops again.

The difference in possible energy savings between 512MB and 1024MB memory node size is apparent. While "mempm base" can save 30% static energy with 512MB nodes, there is no difference with 1024MB nodes in the long run. With migration enabled, our modification is still able to save between 15-20%, however.

Migrate on Fault

Figure 5.9 shows the effect of migration on fault. We compare the version without migration, the basic migration without migrate on fault and the final version with migrate on fault enabled. The version with migration, but without migrate on fault, initially performs a number of migrations. The persistent load, however, results in many late accesses that re-activate the source node. The policy enters the exponential back-off and does not attempt further migrations. In the long run, the basic migration performs similar to the version without migration. In contrast to this, the version with migrate on fault manages to actually reduce the active–active node set. The effect is most evident for the migration at around 250 seconds. The policy performs a rather large migration—there is a noticeable spike in the energy consumption. For the next

250 seconds after the migration, there are a number of late accesses and there is no immediate drop in the energy consumption. Finally, the late accesses stop, the node becomes truly idle, and the energy consumption drops due to the reduced active–active node set. All other tests herein that mention migration were performed with migrate on fault activated.

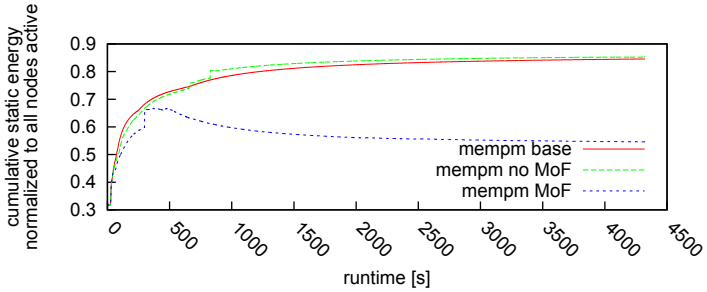


Figure 5.9: A single FreeBSD guest with 4GB of virtual main memory running the NetBSD source compilation 20 times without sleeping between runs on 512MB nodes

5.5.3 Workloads with Multiple Guests

Our design and implementation focus on a single processor environment. At every point of time only one guest is active and only the active guest causes memory accesses. In addition, our changes ensure that the memory energy used by each individual guest are optimized to reduce energy. Thus, we do not expect a different outcome from running more than one guest at a time. In this section we show that this is the case by running three guests on the same processor. We choose the FreeBSD and Debian kernel compilation, together with the SPEC benchmark guest. We did not include the OpenSolaris guest in these tests as OpenSolaris is unable to really shut down the guest, which makes automated testing difficult.

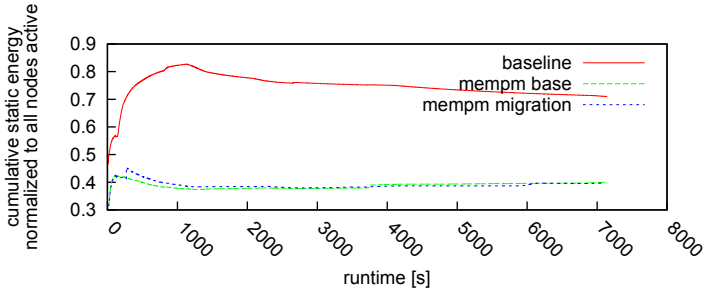


Figure 5.10: FreeBSD (1GB), SPEC (2GB), and Debian (1GB) with 512MB nodes

Figures 5.10 and 5.11 show the plots for this test. As expected, the result is an average of the results provided above. The spike in the energy consumption of “mempm migration” at the beginning of the trace is due to migration in the FreeBSD guest. It takes longer for this migration to pay off as the resulting benefit is only actualized while the FreeBSD guest is active, but the absolute cost of the migration is accounted for directly. Relative to the runtime of the FreeBSD guest, the situation remains the same as above.

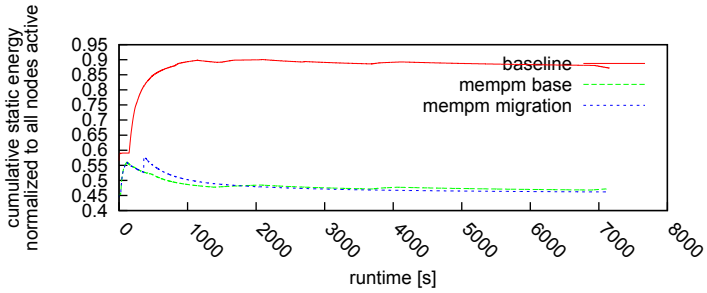


Figure 5.11: FreeBSD (1GB), SPEC (2GB), and Debian (1GB) with 1024MB nodes

Chapter 6

Conclusion

In this thesis, we have presented a design for memory management of virtual machines with the goal to save energy in the memory system. We have adapted a previously proposed allocation policy—designed for traditional processes in an operating system—to the specific challenges of the virtualization environment. We identified different types of memory that are in use on a virtualization platform and discussed special ways to handle the allocation for each memory type. In addition, we have described a system to track the working set of a virtual machine and a policy that uses this information to improve the initial placement by migrating active memory to a better location. To our knowledge, this is the first work to investigate memory power-management in the specific environment of virtualization.

With a prototypical implementation of our allocation policy, the working set detection, and the migration policy, we were able to show that our system can save up to 55% of the static energy consumption in the memory system compared to a simple, reactive hardware power-management. We also demonstrated that our migration policy can help to save additional energy in some scenarios.

In addition we have identified similar requirements between our design and the problem to support NUMA systems. As NUMA systems are on the rise in the consumer market, we believe that the basic mechanisms to support a power aware allocation policy will be available on a large number of hypervisors. It is our hope that this synergy effect will help to promote solutions, such as ours, in future hypervisor implementations—once functioning hardware power-management for the memory system becomes available, again.

Our design also considered additional opportunities to save energy, which we did not implement. We discussed the impact of the scheduling on the energy consumption. Future work can use our findings to implement improved scheduling policies based on our working set detection. We excluded the use of page sharing as a viable way to save energy in the memory subsystem, but found that this technique can help to save energy in the system as a whole. Future work should investigate the interactions between memory and secondary storage and balance the energy savings in either system. We also elaborated on the use of para-virtualization techniques to improve our design. Future work should investigate possible implementations of the principle ideas presented in our work. Finally, we outlined our thoughts on multi-processor host systems and the special challenges that these system present for the scheduling. Future work in this area should combine our design with other work that is concerned with resource based scheduling. Our working set detection can help to provide the required resource

information.

We feel that the main shortcoming of this work is the lack of real measurements. Our evaluation is purely based on a memory energy model. We plan to conduct additional experimentation on a different hardware setup that allows us to further evaluate our design. We are currently looking for a platform that can either provide functioning hardware memory power-management, or a system that can trace memory bus accesses without overhead.

Bibliography

[1] AMD. Amd-v nested paging. Technical report, Advanced Micro Devices, Inc., 2008.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, and Rolf Neugebauer. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, Bolton Landing, NY, USA, October 2003. ACM.

[3] Victor Delaluz, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Anand Sivasubramaniam, and Mary Jane Irwin. DRAM energy management using software and hardware directed power mode control. In *HPCA*, pages 159–170, 2001.

[4] Victor Delaluz, Anand Sivasubramaniam, Mahmut T. Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. Scheduler-based DRAM energy management. In *Proceedings of the 39th Design Automation Conference (DAC 2002)*, pages 697–702, New York, June 10–14 2002. ACM Press.

[5] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Memory controller policies for DRAM power management. In *Proceedings of the 2001 International Symposium on Low-Power Electronics and Design (ISLPED'01)*, August 2001.

[6] Justin M. Forbes. Ksm: Kernel shared memory for linux.

[7] Darryl Gove. Cpu2006 working set size. *SIGARCH Comput. Archit. News*, 35(1):90–96, 2007.

[8] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[9] Hai Huang, Kang G. Shin, Charles Lefurgy, Karthick Rajamani, Tom W. Keller, Eric Van Hensbergen, and Freeman L. Rawson III. Software-hardware cooperative power management for main memory. In Babak Falsafi and T. N. Vijaykumar, editors, *PACS*, volume 3471 of *Lecture Notes in Computer Science*, pages 61–77. Springer, 2004.

[10] Rambus Inc. Rdram memory architecture http://www.rambus.com/us/products/rdram/index.html.

[11] D. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA., 1973.

[12] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Schlatter Ellis. Power aware page allocation. In *ASPLOS*, pages 105–116, 2000.

[13] Min Lee, Euiseong Seo, Joonwon Lee, and Jinsoo Kim. PABC: Power-aware buffer cache management for low power consumption. *IEEE Trans. Computers*, 56(4):488–501, 2007.

[14] Andreas Merkel and Frank Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. In Joseph S. Sventek and Steven Hand, editors, *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 1–12. ACM, 2008.

[15] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In *Usenix*, 2009.

[16] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[17] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. 3rd Int'l. Conf. on Distr. Computing Sys.*, page 22, October 1982.

[18] Vivek Pandey, Weihang Jiang, Yuanyuan Zhou, and Ricardo Bianchini. DMA-aware memory energy management. In *HPCA*, pages 133–144. IEEE Computer Society, 2006.

[19] Samsung Electronics. *DDR3 SDRAM Specification*.

[20] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI*, 2002.

[21] Andreas Weissel, Bjoern Beutel, and Frank Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)*, December 2002.

[22] Wikipedia. Hardware performance counter — wikipedia, the free encyclopedia, 2009. [Online; accessed 18-November-2009].

[23] Wikipedia. Kernel-based virtual machine — wikipedia, the free encyclopedia, 2009. [Online; accessed 20-November-2009].

[24] Wikipedia. Linux — wikipedia, the free encyclopedia, 2009. [Online; accessed 20-November-2009].

[25] Wikipedia. Qemu — wikipedia, the free encyclopedia, 2009. [Online; accessed 20-November-2009].

[26] Wikipedia. Standard performance evaluation corporation — wikipedia, the free encyclopedia, 2009. [Online; accessed 18-November-2009].

[27] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *Proc. 13th International Conference on High-Performance Computer Architecture (13th HPCA'07)*, pages 306–317, San Francisco, CA, USA, February 2007. IEEE Computer Society.

[28] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. *ACM SIGPLAN Notices*, 39(11):177–188, November 2004.