Leveraging MPST in Linux with Application Guidance to Achieve Power and Performance Goals

Michael R. Jantz
University of Kansas
mjantz@ittc.ku.edu

Kshitij A. Doshi
Intel Corporation
kshitij.a.doshi@intel.com

Prasad A. Kulkarni
University of Kansas
kulkarni@ittc.ku.edu

Heechul Yun
University of Kansas
heechul@ittc.ku.edu

Abstract

In this work, we describe an approach that improves collaboration between applications, the Linux kernel, and hardware memory subsystem (controllers and the DIMMs) in order to balance power and performance objectives, and we present details of its implementation using the Linux 2.6.32 kernel (x64) as base. The implementation employs ACPI memory power state table (MPST) to organize system memory into power domains according to rank information. An application programming interface (API) in our implementation allows applications to efficiently communicate various provisioning goals concerning groups of virtual ranges to the kernel. The kernel biases allocation and reclamation algorithms in line with the provisioning goals. The goals may vary over time; thus at one time, the applications may request high power efficiency; and at another time, they may ask for bandwidth or capacity reservations, and so on. This paper describes the framework, the changes for incorporating MPST information, policy modifications, and examples and use cases for invoking the new capabilities.

1 Introduction

Recent computing trends necessitate an increased focus on power and energy consumption and support for multi-tenant use cases. There is therefore a need to multiplex hardware efficiently and without performance interference. Advances in allocating CPU, storage and network resources have made it possible to meet competing service quality objectives while reducing power or energy demands[9, 8, 3]. In comparison to other resources, however, it is very challenging to obtain precise control over distribution of memory capacity, bandwidth, or power, when virtualizing and multiplexing system memory. Precisely controlling memory power and performance is difficult because these effects intimately depend upon the results of activities across multiple layers of the vertical execution stack, which are often not available at any single layer or component.

In an effort to simplify resource management within each layer, current systems often separate and abstract away information necessary to coordinate cross-layer activity. For example, the Linux kernel views physical memory as a single, large, contiguous array of physical addresses. The physical arrangement of memory modules, and that of the channels connecting them to processors, together with the power control domains are all opaque to the operating system's memory management routines. Without exposing this information to the upper-level software, it is very difficult to design schemes that coordinate application demands with the layout and architecture of the memory hardware.

The selection of physical pages to bind to application virtual addresses also has a significant impact on memory power and performance. Operating systems use heuristics that reclaim either the oldest, or the least recently touched, or least frequently used physical pages in order to fill demands. Over time, after repeated allocations and reclaims, there is no guarantee that a collection of intensely accessed physical pages would remain confined to a small number of memory modules (or DIMMs). Even if an application reduces its dynamic memory footprint, its memory accesses can remain spread out across sufficiently many memory ranks to keep any ranks from transitioning into a low-power state to save power. The layout and distribution of each

application's hot pages not only affects the ability of memory modules to transition to lower power states during intervals of low activity, but also impacts the extent of interference caused by a program's activity in memory and the responsiveness experienced by other active programs. Thus a more discriminating approach than is available in current systems for multiplexing of physical memory is highly desirable.

Furthermore, data-intensive computing continues to raise demands on memory. Recent studies have shown that memory consumes up to 40% of total system power in enterprise servers [7] making memory power a dominant factor in overall power consumption. If an applications high-intensity accesses are concentrated among a small fraction of its total address space, then it is possible to achieve power-efficient performance by co-locating the active pages among a small fraction of DRAM banks. At the same time, an application that is very intensive in its memory accesses may prefer that pages in its virtual address span are distributed as widely as possible among independent memory channels to maximize performance. Thus, adaptive approaches are needed for improving power efficiency and performance isolation in the scheduling of memory.

We have designed and implemented a Linux kernel-based framework that improves collaboration between the applications, Linux kernel, and memory hardware in order to provide increased control over the distribution of memory capacity, bandwidth, and power. The approach employs the ACPI memory power state table (MPST)[1], which specifies the configuration's memory power domains and their associated physical addresses. Our modified Linux kernel leverages this information to organize physical memory pages into software structures (an abstraction called "trays") that capture the physically independent power domains. The modified kernel's page management routines perform all allocation and recycling over our software trays.

Our framework includes an application programming interface (API) that allows applications to efficiently communicate provisioning goals to the kernel by applying *colors* to portions of their virtual address space. A color is simply a hint applied to a virtual address range that indicates to the operating system that some common behavior or intention spans pages, even if the pages are not virtually contiguous. Applications can also associate *attributes* (or combinations of attributes) with each color. Attributes provide information (typically some intent or provisioning goal) to the operating system about how to manage the colored range. Colors and their associated attributes can be applied and changed at any time, and our modified Linux kernel attempts to interpret and take them into account while performing allocation, recycling, or page migration decisions.

We re-architect the memory management of a recent Linux kernel (x64, version 2.6.32) to implement our approach. Our recently published work, *A Framework for Application Guidance in Virtual Memory Systems* [6], describes the high-level design and intuition behind our approach and presents several experiments showing how it can be used to achieve various objectives, such as power savings and capacity provisioning. In this work, we provide further design and implementation details, including major kernel modifications and specific functions and tools provided by our coloring API.

The next section describes how our custom kernel leverages MPST information to construct trays and provides details of the structural and procedural modifications necessary to perform allocation and recycling over trays. In Section 3, we present our application programming interface to efficiently communicate application intents to our Linux kernel using colors, and we provide details of the kernel modifications that are necessary to receive and interpret this communication. In Section 4 we discuss our plans for future work, and Section 5 concludes the paper.

2 Leveraging MPST in the Linux Kernel

Our custom kernel organizes physical memory pages into the power-manageable tray abstraction by leveraging information provided by the ACPI memory power state table. Our implementation enables tray-based memory allocation and reclaim policies, which we describe in the following section.

2.1 Tray Design

Modern server systems employ a Non-Uniform Memory Access (NUMA) architecture which divides memory into separate regions (*nodes*) for each processor or set of processors. Within each NUMA node, memory is spatially organized into *channels*. Each channel employs its own memory controller and contains one or more DIMMs, which, in turn, each contain two or

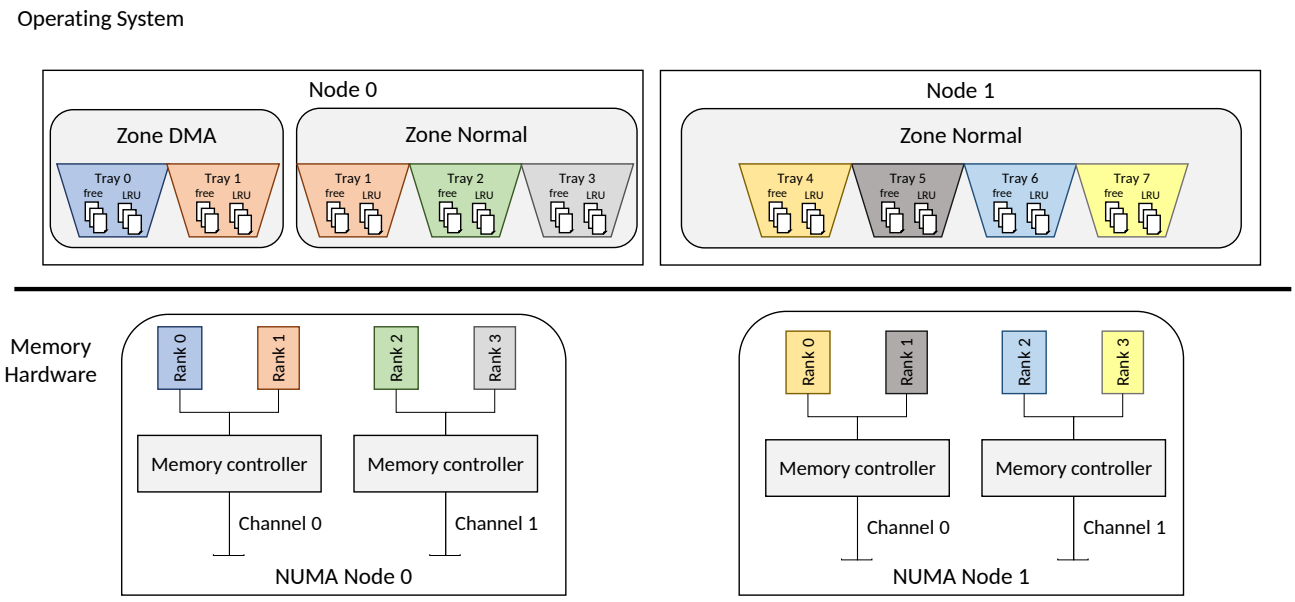

Figure 1: Organization of tray structures in relation to memory hardware

more *ranks*. Ranks comprise the actual memory storage and typically range from 2GB to 8GB in capacity. The memory hardware performs aggressive power management to transition from high power to low power states when either all or some portion of the memory is not active. Ranks are the smallest *power manageable unit*, which implies that transitioning between power states is performed at the rank level. Thus, different memory allocation strategies must consider an important power-performance tradeoff: distributing memory evenly across the ranks improves bandwidth which leads to better performance, while minimizing the number of active ranks consume less power.

The Linux kernel maintains a hierarchy of structures to represent and manage physical memory. *Nodes* in the Linux kernel correspond to the physical NUMA nodes in the hardware. Each node is divided into a number of blocks called *zones*, which represent distinct physical address ranges. At boot time, the OS creates physical page frames (or simply, *pages*) from the address range covered by each zone. Each page typically addresses 4KB of space. The kernel's physical memory management (allocation and recycling) operates on these pages, which are stored and tracked using various lists in each zone. For example, a set of lists of pages in each zone called the *free lists* describes all of the physical memory available for allocation.

To implement our approach, we create a new division in this hierarchy called *trays*. A tray is a software structure which contains sets of pages that reside on the same power-manageable memory unit. Each zone contains a set of trays and all the lists used to manage pages on the zone are replaced with corresponding lists in each tray. Figure 1 shows how our custom kernel organizes its representation of physical memory with trays in relation to the actual memory hardware.

One potential issue with the tray approach is that power-manageable domains that cross existing zone boundaries are represented as separate trays in different zones. Tray 1 in Figure 1 illustrates this situation. Another approach, proposed by A. Garg [5], introduces a *memory region* structure between the node and zone level to capture power-manageable domains. Although this approach is able to represent power-manageable domains that cross existing zone boundaries in a single structure, it requires a duplicate set of zone structures for each memory region. There are a number of important memory management operations that occur at the zone level and maintaining duplicate zone structures significantly complicates these operations. Our tray-based approach avoids zone duplication, and thus, avoids such complications. A more recent version of the memory region approach, proposed by S. Bhat [4], removes memory regions from the node-zone hierarchy entirely, and captures power-manageable domains in a data structure parallel to zones.

```
Flags (decoded below)     : 03
Node Enabled              : 1
Power Managed             : 1
Hot Plug Capable          : 0
Reserved                  : 00
Node ID                   : 0000
Length                    : 00000026
Range Address             : 0000000700000000
Range Length              : 000000013FFFFFFF
Num Power States          : 02
Num Physical Components   : 03
Reserved                  : 0000
...
```

Figure 2: Example MPST entry

2.2 Mapping Pages to Trays Using MPST

Assigning pages to the appropriate tray requires a mapping from physical addresses to the power-manageable units in hardware. We employ the ACPI defined memory power state table (MPST), which provides this mapping. Each entry in the MPST specifies a memory power node with its associated physical address ranges and supported memory power states. Some nodes may have multiple entries to support nodes mapped to non-contiguous address ranges.

The BIOS presents this information to the kernel at boot time in the form of an ACPI Data Table with statically packed binary data. Unfortunately, our kernel version does not include any facilities to parse the MPST into structured data. Therefore, we parse the table into text using utilities provided by the ACPI Component Architecture (ACPICA) project [2]. Figure 2 shows an example MPST entry as text. The most important fields in this table for defining power-manageable domains are `Range Address` and `Range Length`, which specify the physical address range of each memory power node. Thus, by either building this information into the kernel image or by copying it from user space during runtime, we are able to construct a global list of memory power nodes and their associated physical address ranges for use during memory management.

Pages can now be assigned to the appropriate tray by searching the global list of memory power nodes to find which node contains each page of memory. For most system configurations, this list is relatively short and searching it does not require significant overhead. Thus, to simplify our implementation effort, our custom kernel performs this search every time an operation needs to determine which tray should contain a particular page. An efficient implementation could perform this search once for each page and store information identifying the page's tray in the page flags field, similar to how the zone and node information for each page are currently stored.

It is important to note that our framework assumes that all of the physical addresses on each page correspond to memory on the same power-manageable unit in hardware. Some configurations interleave physical addresses within each page across power-manageable units in the hardware. For example, in an effort to exploit spatial locality, some systems interleave physical addresses across channels at the cache line boundary (typically 64 bytes). Supposing such a system has two DIMMs, each connected to its own channel, the first cache line within a page will use DIMM 0, the next will use DIMM 1, the next will use DIMM 0, and so on. In this case, our framework cannot control access to each individual DIMM, but may be able to control access to which ranks are used within the DIMMs.

2.3 Memory Management Over Trays

To enable memory management over trays, we modified our kernel's page management routines, which operate on lists of pages at the zone level, to operate over the same lists, but at a subsidiary level of trays. That is, zones are subdivided into trays, and page allocation, scanning, recycling are all performed at the tray level. For example, during page allocation, once a zone that is sufficient for a particular allocation has been found, the allocator calls `buffered_rmqueue` to find a page and remove it from the zone's free lists. In our custom kernel, `buffered_rmqueue` is modified to take an additional argument describing which tray's free lists should be searched, and each call site is modified to call `buffered_rmqueue` repeatedly, with each tray, until a suitable page is found. Most of the other required changes are similarly straightforward.

One notable complication has to do with the low-level page allocator in Linux, known as the buddy allocator. In order to quickly fulfill requests for contiguous ranges of pages, the Linux buddy allocator automatically groups contiguous blocks of pages. The *order* of a block of pages refers to the number of contiguous

```
struct tray {
  ...
  /*
   * free lists of pages of different orders
   */
  struct free_area    free_area[MAX_ORDER];
  ...
};
struct free_area {
  struct list_head    free_list[MIGRATE_TYPE];
  unsigned long       nr_free;
};
```

Figure 3: Definition of free lists in the buddy system

pages in that block, where an order-n block contains 2^n contiguous pages. Block orders range from 0 (individual pages) to some pre-defined maximum order (11 on our Linux 2.6.32 x64 system). Blocks of free pages are stored in a set of lists, where each list contains blocks of pages of the same order. In our implementation, the lists are defined at the tray-level as shown in Figure 3. When two contiguous order-n blocks of pages are both free, the buddy allocator removes these two blocks from the order-n free list and creates a new block to store on the order-$n+1$ free list. However, if two contiguous order-n blocks reside in different trays, their combined order-$n+1$ block cannot be placed on either tray (as it would contain pages belonging to the other tray). For this reason, we maintain a separate set of free lists, defined at the zone level, to hold higher order blocks that contain pages from separate trays. With this structure, our custom kernel is able to fulfill requests for low-order allocations from a particular power-manageable domain, while, at the same time, it is also able to handle requests for large blocks of contiguous pages, which may or may not reside on the same power-manageable unit.

3 Application Guidance under Linux

The goal of our framework is to provide control over memory resources such as power, bandwidth, and capacity in a way that allows the system to flexibly adapt to shifting power and performance objectives. By organizing physical memory into power-manageable domains, our kernel patch provides crucial infrastructure for enabling fine-grained resource management in software. However, memory power and performance depend not only on how physical pages are distributed across the memory hardware, but also on how the operating system binds virtual pages to physical pages, and on the demands and usage patterns of the upper-level applications. Thus, naïve attempts to manage these effects are likely to fail.

Our approach is to increase collaboration between the applications and operating system by allowing applications to communicate how they intend to use memory resources. The operating system interprets the application's intents and uses this information to guide memory management decisions. In this section, we describe our *memory coloring* interface that allows applications to communicate their intents to the OS, and the kernel modifications necessary to receive, interpret, and implement application intents.

3.1 Memory Coloring Overview

A color is an abstraction which allows the application to communicate to the OS hints about how it is going to use memory resources. Colors are sufficiently general as to allow the application to provide different types of performance or power related usage hints. In using colors, application software can be entirely agnostic about how virtual addresses map to physical addresses and how those physical addresses are distributed among memory modules. By coloring any N different virtual pages with the same color, an application communicates to the OS that those N virtual pages are alike in some significant respect, and by associating one or more attributes with that color, the application invites the OS to apply any discretion it may have in selecting the physical page frames for those N virtual pages.

By specifying coloring hints, an application provides a usage map to the OS, and the OS consults this usage map in selecting an appropriate physical memory scheduling strategy for those virtual pages. An application that uses no colors and therefore provides no guidance is treated normally – that is, the OS applies some default strategy. However, when an application provides guidance through coloring, depending on the particular version of the operating system, the machine configuration (such as how much memory and how finely interleaved it is), and other prevailing run time conditions in the machine, the OS may choose to veer a little or a lot from the default strategy.

System Call	Arguments	Description
`mcolor`	`addr`, `size`, `color`	Applies `color` to a virtual address range of length `size` starting at `addr`
`get_addr_mcolor`	`addr`,`*color`	Returns the current color of the virtual address `addr`
`set_task_mcolor`	`color`	Applies `color` to the entire address space of the calling process
`get_task_mcolor`	`*color`	Returns the current color of the calling process' address space
`set_mcolor_attr`	`color`, `*attr`	Associates the attribute `attr` with `color`
`get_mcolor_attr`	`color`, `*attr`	Returns the attribute currently associated with `color`

Table 1: System calls provided by the memory coloring API

3.2 Memory Coloring Example

The application interface for communicating colors and intents to the operating system uses a combination of configuration files, library software, and system calls. Let us illustrate the use of our memory coloring API with a simple example.

Suppose we have an application that has one or more address space extents in which memory references are expected to be relatively infrequent (or uniformly distributed, with low aggregate probability of reference). The application uses a color, say *blue* to color these extents. At the same time, suppose the application has a particular small collection of pages in which it hosts some frequently accessed data structures, and the application colors this collection *red*. The coloring intent is to allow the operating system to manage these sets of pages more efficiently – perhaps it can do so by co-locating the *blue* pages on separately power-managed units from those where *red* pages are located, or, co-locating *red* pages separately on their own power-managed units, or both. A possible second intent is to let the operating system page the *blue* ranges more aggressively, while allowing pages in the *red* ranges an extended residency time. By locating *blue* and *red* pages among a compact group of memory ranks, an operating system can increase the likelihood that memory ranks holding the *blue* pages can transition more quickly into self-refresh, and that the activity in *red* pages does not spill over into those ranks. Since many usage scenarios can be identified to the operating system, we define "intents" and specify them using configuration files. A configuration file for this example is shown in Figure 4. In this file, the intents labeled `MEM-INTENSITY` and `MEM-CAPACITY` can capture two intentions: (a) that red pages are hot and blue pages are cold, and (b) that about 5% of application's dynamic resident set size (RSS) should fall into red pages, while, even though there are many blue pages, their low probability of access is indicated by their 3% share of the RSS. Next, let us examine exactly how these colors and intents are actually communicated to the operating system using system calls.

```
# Specification for frequency of reference:
INTENT   MEM-INTENSITY

# Specification for containing total spread:
INTENT   MEM-CAPACITY

# Mapping to a set of colors:
MEM-INTENSITY RED   0 //hot pages
MEM-CAPACITY  RED   5 //hint - 5% of RSS

MEM-INTENSITY BLUE 1 //cold pages
MEM-CAPACITY  BLUE 3 //hint - 3% of RSS
```

Figure 4: Example config file for colors and intents

3.3 System Calls in the Memory Coloring API

The basic capabilities of actually applying colors to virtual address ranges and binding attributes to colors are provided to applications as system calls. Applications typically specify colors and intents using configuration files as shown in the example above, and then use a library routine to convert the colors and intents into system call arguments at runtime.

Table 1 lists and describes the set of system calls provided by our memory coloring API. To attach colors to virtual addresses, applications use either `mcolor` (to color a range of addresses), or `set_task_mcolor` (to color their entire address space). Colors are represented as unique integers, and can be overlapped. Thus, we use an integer bit field to indicate the set of colors that have been applied to each address range. To implement `mcolor`, we add a color field to the `vm_area_struct`

```
struct mcolor_attr {
  unsigned int  intent[MAX_INTENTS];
  unsigned int  mem_intensity;
  float         mem_capacity;
};
```

Figure 5: Example attribute structure definition

kernel structure. In the Linux kernel, each process' address space is populated by a number of regions with distinct properties. These regions are each described by an instance of `vm_area_struct`. When an application calls `mcolor`, the operating system updates the `vm_area_struct` that corresponds to the given address range (possibly splitting an existing `vm_area_struct` and/or creating a new `vm_area_struct`, if necessary) to indicate that the region is colored. The `set_task_mcolor` implementation is similar: the `task_struct` kernel structure is modified to include a color field, and `set_task_mcolor` updates this field. In the case that the `vm_area_struct` and `task_struct` color fields differ, the operating system may attempt to resolve any conflicts. In our default implementation, we simply choose the color field on the `vm_area_struct` if it has been set.

Colors are associated with attributes using the `set_mcolor_attr` system call. To represent attributes, we use a custom structure called `mcolor_attr`. For each color, this structure packages all of the data necessary to describe the color's associated intents. Figure 5 shows the `mcolor_attr` definition that is used for the example in Section 3.2. The `intent` field indicates whether a particular type of intent has been specified, and the remaining fields specify data associated with each intent. The kernel maintains a global table of colors and their associated attributes. When the application calls `set_mcolor_attr`, the kernel links the given attribute to the color's position in the global table. Note that in this implementation, there is only one attribute structure associated with each color. We bind multiple intents to one color by packaging the intents together into a single attribute structure.

3.4 Interpreting Colors and Intents During Memory Management

Lastly, we examine exactly how our modified Linux kernel steers its physical memory management decisions in accordance with memory coloring guidance. To describe this process, let us again consider the example in Section 3.2. Before the example application applies any colors, the system employs its default memory management strategy for all of the application's virtual pages. After applying the *red* and *blue* colors and binding these to their associated intents, the system will eventually fault on a colored page. Early during the page fault handling, the system determines the color of the faulting page (by examining the color field on the page's `vm_area_struct`) and looks up the color's associated attribute structure in the global attribute table. Now, the OS can use the color and attribute information to guide its strategy when selecting which physical page to choose to satisfy the fault.

For example, in order to prevent proliferation of frequently accessed pages across many power-manageable units, the operating system might designate one or a small set of the power-manageable domains (i.e. tray software structures) as the only domains that may be used to back *red* pages. Then, when the system faults on a page colored *red*, the OS will only consider physical pages from the designated set of domains to satisfy the fault. As another example, let us consider how the system might handle the `MEM-CAPACITY` intent. When the OS determines that pages of a certain color make up more than some percentage of the application's current RSS, then the system could choose to recycle frames containing pages of that color in order to fill demands. In this way, the system is able to fill demands without increasing the percentage of colored pages in the resident set.

Note that the specializations that an OS may support need not be confined just to selection of physical pages to be allocated or removed from the application's resident set. The API is general enough to allow other options such as whether or not to fault-ahead or to perform read-aheads or flushing writes, or whether or not to undertake migration of active pages from one set of memory banks to another in order to squeeze the active footprint into fewest physical memory modules. In this way, an OS can achieve performance, power, I/O, or capacity efficiencies based on guidance that application tier furnishes through coloring.

4 Future Work

While our custom kernel and API enable systems to design and achieve flexible, application-guided, power-aware management of memory, we do not yet have an understanding of what sets of application guidance and memory power management strategies will be most useful for existing workloads. Furthermore, our current API requires that all coloring hints be manually inserted into source code and does not provide any way to *automatically* apply beneficial application guidance. Thus, we plan to develop a set of tools to profile, analyze, and automatically control memory usage for applications. Some of the capabilities we are exploring include: (a) a set of tools and library software for applications to query detailed memory usage statistics for colored regions, (b) on-line techniques that adapt memory usage guidance based on feedback from the OS, and (c) integration with compiler and runtime systems to automatically partition and color the application's address space based on profiles of memory usage activity.

Simultaneously, we plan to implement color awareness in selected open source database, web server, and J2EE software packages, so that we can exercise complex, multi-tier workloads at the realistic scale of server systems with memory outlays reaching into hundreds of gigabytes. In these systems, memory power can reach nearly half of the total machine power draw, and therefore they provide an opportunity to explore dramatic power and energy savings from application-engaged containment of memory activities. We also plan to explore memory management algorithms that maximize performance by biasing placement of high value data so that pages in which performance critical data resides are distributed widely across memory channels. We envision that as we bring our studies to large scale software such as a complex database, we will inevitably find new usage cases in which applications can guide the operating system with greater nuance about how certain pages should be treated differently from others.

5 Conclusions

There is an urgent need for computing systems that are able to multiplex memory resources efficiently while also balancing power and performance tradeoffs. We have presented the design and implementation of a Linux-based approach that improves collaboration between the application, operating system, and hardware layers in order to provide a fine-grained, flexible, power-aware provisioning of memory. The implementation leverages the ACPI memory power state table to organize the operating system's physical memory pages into power-manageable domains. Additionally, our framework provides an API that enables applications to express a wide range of provisioning goals concerning groups of virtual ranges to the kernel. We have described the kernel modifications to organize and manage physical memory in software structures that correspond to power-manageable units in hardware. Finally, we have provided a detailed description of our API for communicating provisioning goals to the OS, and we have presented multiple use cases of our approach in the context of a realistic example.

References

[1] Advanced configuration and power interface specification, 2011. http://www.acpi.info/spec.htm.

[2] Acpi component architecture (acpica), 2013. http://www.acpi.info/spec.htm.

[3] Vlasia Anagnostopoulou, Martin Dimitrov, and Kshitij A. Doshi. Sla-guided energy savings for enterprise servers. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 120–121, 2012.

[4] Srivatsa S. Bhat. mm: Memory power management, 2013. http://lwn.net/Articles/546696/.

[5] Ankita Garg. mm: Linux vm infrastructure to support memory power management, 2011. http://lwn.net/Articles/445045/.

[6] Michael R. Jantz, Carl Strickland, Karthik Kumar, Martin Dimitrov, and Kshitij A. Doshi. A framework for application guidance in virtual memory systems. In *VEE '13: Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, March 2013.

[7] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, December 2003.

[8] Lanyue Lu, P.J. Varman, and K. Doshi. Decomposing workload bursts for efficient storage resource management. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):860 –873, may 2011.

[9] H. Wang, K. Doshi, and P. Varman. Nested qos: Adaptive burst decomposition for slo guarantees in virtualized servers. *Intel Technology Journal*, June, 16:2 2012.